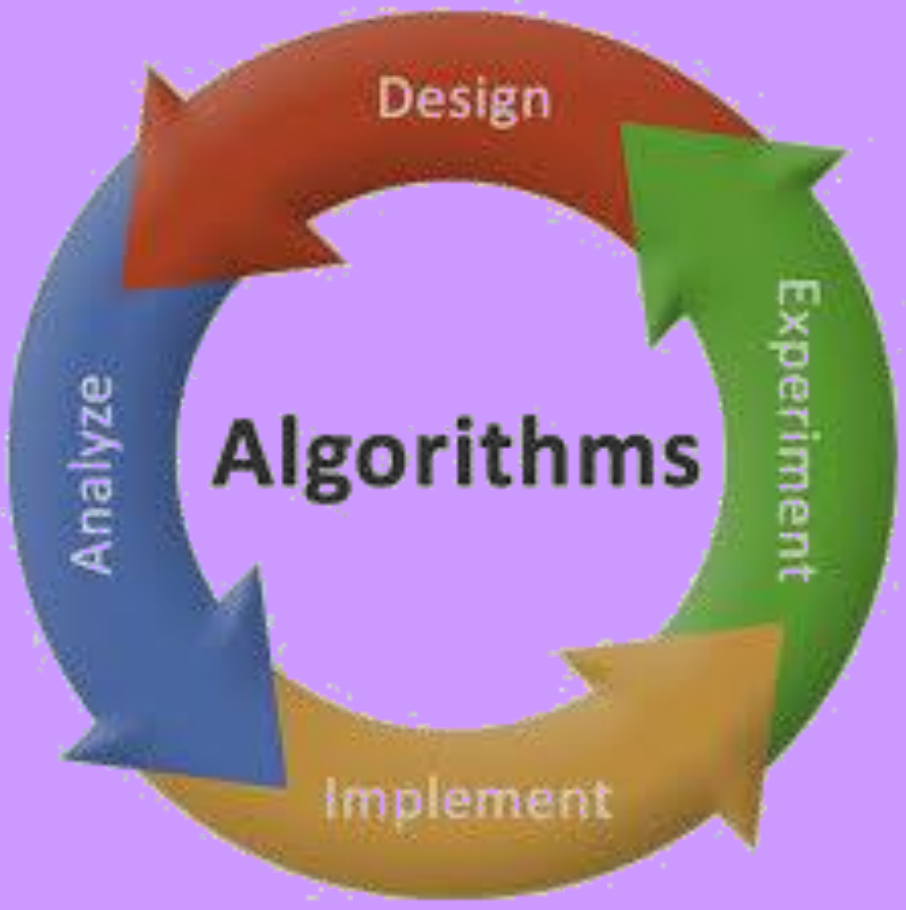
THE MAGICAL GUIDE TO ALGORITHM ANALYSIS AND DESIGN



Rosina S Khan

Dedicated to You,

The Valued Reader

Copyright Information

https://rosinaskhan.weebly.com

Table of Contents

Preface

I have named this guide as "The Magical Guide to Algorithm Analysis and Design" because the very process of writing down a set of instructions in the form of pseudocodes before feeding them into program structures and executing them successfully to get program outputs is very magical indeed. It all starts with the human thinking process and via pseudocodes or algorithms, converting them into programs, including their analysis and design, is nothing short of magic and therefore, the title of this guide.

Analysis is the measurement of the quality of your design. Just like you use your sense of taste to check your cooking, you should get into the habit of using algorithm analysis to justify design decisions when you write an algorithm or a computer program. This is a necessary step to reach the next level in mastering the art of programming.

I have integrated the content of this book from various resources including several google searches. The main titles I have used for this guide are the following:

1) Lecture Notes for Data Structures and Algorithms by John Bullinaria

2) A Handout on Introduction to Data Structures and Algorithms by IDOL

3) Lecture Notes on Algorithm Analysis and Design by Herbert Edelsbrunner

4) A Handout on Introduction to Algorithms by Jon Kleinberg and Eva Tardos

5) A Common-Sense Guide to Data Structures and Algorithms by Jay Wengrow, 2nd Edition, The Pragmatic Programmers

The guide, authored by me, is meant for undergraduate students in the field of Computer Science and Engineering or an equivalent program.

This book is organized into 11 chapters.

In Chapter 1, we introduce the concept of Algorithms and the fundamental questions about algorithms. [1]

Chapter 2 portrays the concept of Data Structures and their varieties such as arrays, linked lists, stacks, and queues. Also, abstract data type and the advantages of abstract data trees are explained. [2, 5]

In Chapter 3, we formally define an algorithm, introduce time and space complexities, the types of analyses of algorithms, and mathematical notations. [2, 5]

Chapter 4 depicts searching algorithms such as linear search and binary search. [1]

In Chapter 5, we discuss the concept of trees both quadtrees and binary trees. [1]

Chapter 6 focuses on binary search trees- how to build and search them, how to sort using them, how to delete nodes from them, and we introduce B-trees. [1]

Chapter 7 deals with the various operations on binary heap trees. [1]

The next chapter, Chapter 8 concentrates on the various sorting techniques such as, Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort, and Heap sort and their algorithmic complexities as well. [2, 5]

Chapter 9 goes on to explain graphs: the basic concepts, terminology used, representations, operations (Depth-First Search and Breadth First Search) and traversals. [2]

Chapter 10 talks about the concepts of spanning tree and minimum spanning tree and their applications, and also, takes into account graph algorithms such as Kruskal's algorithm and Prim's algorithm based on the above concepts. [2, 5]

Chapter 11 is concerned with the theoretical aspects of various Algorithm Design Techniques such as, Divide and Conquer, Backtracking, Dynamic Programming and Greedy Methods. [2, 3, 4, 5]

At the end of the guide, I cater to further free resources, which you will find both valuable and enjoyable.

Acknowledgements

I am extremely grateful to John Bullinaria, IDOL, Herbert Edelsbrunner, Jon Kleinberg and Eva Tardos, and Jay Wengrow for using some of their resources and merging with what I have and therefore, the creation of this guide.

Last but not the least, I am thankful to my family for their support during the write-up of this eBook.

C H A P T E R 1

Introduction

1. Introduction to Algorithms

The following chapters cover the key ideas involved in designing algorithms. We shall see how they depend on the design of suitable data structures, and how some structures and algorithms are more efficient than others for the same task. We will concentrate on a few basic tasks, such as storing, sorting, and searching data, that underlie much of computer science, but the techniques discussed will be applicable much more generally.

Throughout, we will investigate the computational efficiency of the algorithms we develop, and gain intuitions about the pros and cons of the various potential approaches for each task. We will not restrict ourselves to implementing the various data structures and algorithms in particular computer programming languages (e.g., Java, C etc.), but specify them in simple pseudocode that can easily be implemented in any appropriate language.

1.1 Algorithms as Opposed to Programs

An algorithm for a particular task can be defined as a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

As such, an algorithm must be precise enough to be understood by human beings. However, in order to be executed by a computer, we will generally need a program that is written in a rigorous formal language; and since computers are quite inflexible compared to the human mind, programs usually need to contain more details than algorithms.

Here, we shall ignore most of those programming details and concentrate on the design of algorithm rather than programs. The task of implementing the discussed algorithms as computer programs is important, of course, but these chapters will concentrate on the theoretical aspects and leave the practical programming aspects to be studied elsewhere.

Having said that, we will often find it useful to write down segments or whole of actual programs in some areas in order to clarify and test certain theoretical aspects of algorithms and their data structures.

Algorithms can obviously be described in plain English, and we will sometimes do that. However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted. This is called pseudocode, which comes in a variety of forms. Often the chapters will present segments of pseudocode that are very similar to the languages we are mainly interested in, namely the overlap of C and Java, with the advantage that they can easily be inserted into runnable programs.

1.2 Fundamental Questions About Algorithms

Given an algorithm to solve a particular problem, we are naturally led to ask:
1. What is it supposed to do?
2. Does it really do what it is supposed to do?
3. How efficiently does it do it?

The technical terms normally used for these three aspects are:
1. Specification.
2. Verification.
3. Performance analysis.

The details of these three aspects will usually be rather problem dependent.

The specification should formalize the crucial details of the problem that the algorithm is intended to solve. Sometimes that will be based on a particular representation of the associated data, and sometimes it will be presented more abstractly. Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous.

For simple problems, it is often easy to see that a particular algorithm will always work i.e., that it satisfies its specification. However, for more complicated specifications and/or algorithms, the fact that an algorithm satisfies its specification may not be obvious at all. In this case, we need to spend some effort verifying whether the algorithm is indeed correct. In general, testing on a few particular inputs can be enough to show that the algorithm is incorrect.

Finally, the efficiency or performance of an algorithm relates to the resources required by it, such as how quickly it will run, or how much computer memory it will use. This will usually depend on the problem instance size, the choice of data representation, and the details of the algorithm. Indeed, this is what normally drives the development of new data structures and algorithms. We shall study the general ideas concerning efficiency and then apply them throughout the remainder of the chapters.

C H A P T E R 2

Data Structures

2. Introduction

What are data structures? They are structured data with logical relationships between data elements. For example, a street address can be identified by street number and street name. These structured data variables depend on one another to create a unique structure, the street address. In data structure, a linked list, for example, would link data elements to form a structured component of the system.

2.1 Basic Terminology

We use some terminology while working with data structures, which should be obvious to every Computer Science undergraduate student. These are explained below:

Data: Data can be defined as a fundamental value, or a collection of values. For example, data about a student can be his student id and name.

Group/Composite Item: A group or composite item has parts or subordinate items. For example, a student's name can have first name, middle name, and last name as parts or subordinate items so that the composite/group item here is the student's name.

Attribute/Entity: An entity has properties or attributes. For example, the student entity has properties or attributes as name, id, address, phone number etc.

Field/Record: The student entity actually becomes a collection of records in a file. For each of them pertaining to a specific student. The fields of a record are the data or attributes of a student for example, id, name, address, phone number.

2.2 The Need for Data Structure

We need a data structure for an organization because:

1) It helps to define the different levels of the organization.
2) It provides a means of storing the data, and also retrieving them at the elementary core.
3) It helps to carry out operations on the stored data such as, deleting, updating, or adding items, or even extracting the highest/lowest priority data item.
4) It helps to store huge amounts of data efficiently.
5) It enables searching and sorting of data conveniently.

2.3 The Goals of Data Structure

1) Whatever problems the organization needs to solve, the data structure does so correctly for all kinds of input.
2) The data structure needs to be efficient as well. It must process the data at a high speed without utilizing much of memory space.
3) Implementation of the data structure may require a certain amount of programming effort.

2.4 Steps in Selecting a Data Structure

1) Analyze your problem to support the basic operations.
2) Quantify the data constraints for the operations involved.
3) Select the data structure that best satisfies the requirements.

The first concern is the data and the data operations. The next concern is the representation of those data, and the final concern is the implementation of the representation.

2.5 Classification of Data Structures

A data structure represents the relationship between data elements and helps programmers to process data easily.

There are mainly two types of data structures:

1) Primitive Data Structures
2) Non-primitive Data Structures

Fig 2.1 shows the different classifications of data structures.

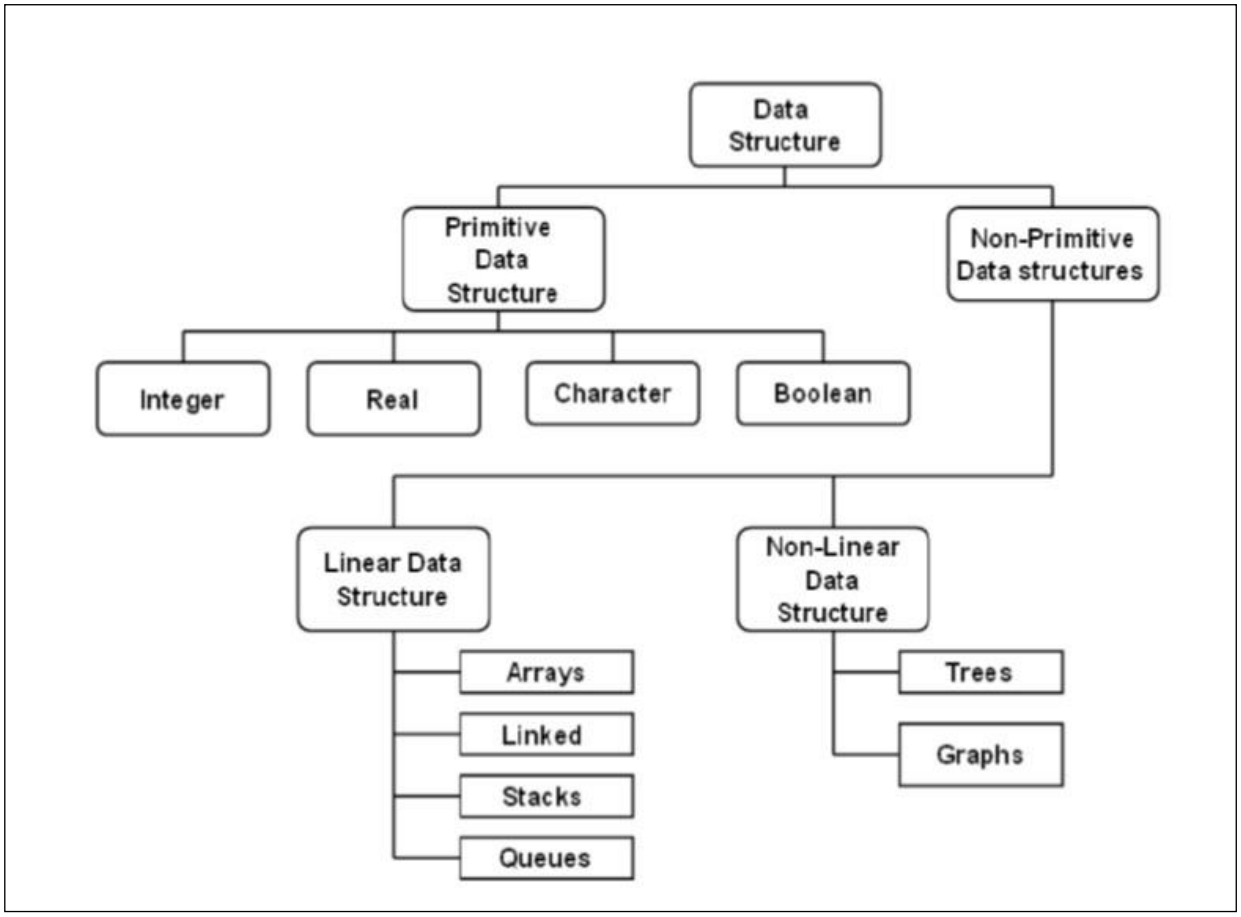


Fig 2.1: Classification of Data Structures

2.6 Primitive Data Structure

Basic data types such as integer, real, character and Boolean fall under primitive data structure category. These data types are simple because they all consist of characters and cannot be further divided. They can be manipulated and operated by machine level instructions.

2.7 Non-Primitive Data Structures

Non-primitive data structures are derived from primitive ones. They are based on data elements that can of the same data type (homogeneous) or different data types (heterogeneous). They cannot be operated by machine level instructions. They can be further divided into linear and non-linear data structures, depending on the structure and arrangement of data.

2.7.1 Linear Data Structures

A data structure that maintains a linear relationship among its data elements is a linear data structure. However, in memory, the data may not be sequential. Examples of these can be array, linked list, stack, queue.

2.7.2 Non-Linear Data Structures

This data structure does not consist of data elements in a linear fashion. Rather they are arranged in a hierarchical arrangement. Insertion and deletion of data items cannot be done here in a linear way. Examples of non-linear data structures are trees and graphs. They will be explained in detail in later chapters.

Array

An array is an orderly arrangement of data elements. These data elements are stored in adjacent locations of the data structure. They are stored linearly with the same data type. So, an array is also called a linear homogeneous data structure.

We can declare an array Arr with 6 values as follows:

int Arr[6]= {56, 17, 60, 9, 7, 10}

This declaration will create an array as shown in the following figure:

Arr	0	1	2	3	4	5
	56	17	60	9	7	10

Fig. 2.2: An Array

Arrays can be classified as one-dimensional, two-dimensional, or multidimensional.

- **One-dimensional Array:** It has only one row of elements. It is stored in ascending storage locations.
- **Two-dimensional Array:** It consists of multiple rows and columns of data elements. It is also called as a matrix.
- **Multidimensional Array:** Multidimensional arrays can be defined as an array of arrays. Multidimensional arrays are not bounded to two indices or two dimensions.
  They can include as many indices as required.

Limitations of Arrays

1) Arrays are of fixed size.
2) Data elements are stored in contiguous memory locations, but they may not always be available.
3) Insertion and deletion of data elements may be problematic because they need to be shifted from their locations.

These limitations can, however, be solved by the use of linked lists.

Applications

1) Storing data elements of the same data type.
2) Auxiliary storage for other data structures.
3) Storage of binary elements of fixed count.
4) Storage of matrices.

Linked List

A linked list is a data structure in which a data element points or links to the next data element of the list. Here data elements need not have consecutive memory locations. Insertion and deletion of data elements are possible anywhere in the linked list. It allocates a block of memory for each data item. For this reason, a linked list is considered a chain of data elements or records called nodes. Each node contains information and pointer fields. The information field contains the actual data while the pointer field contains a pointer to the next node.

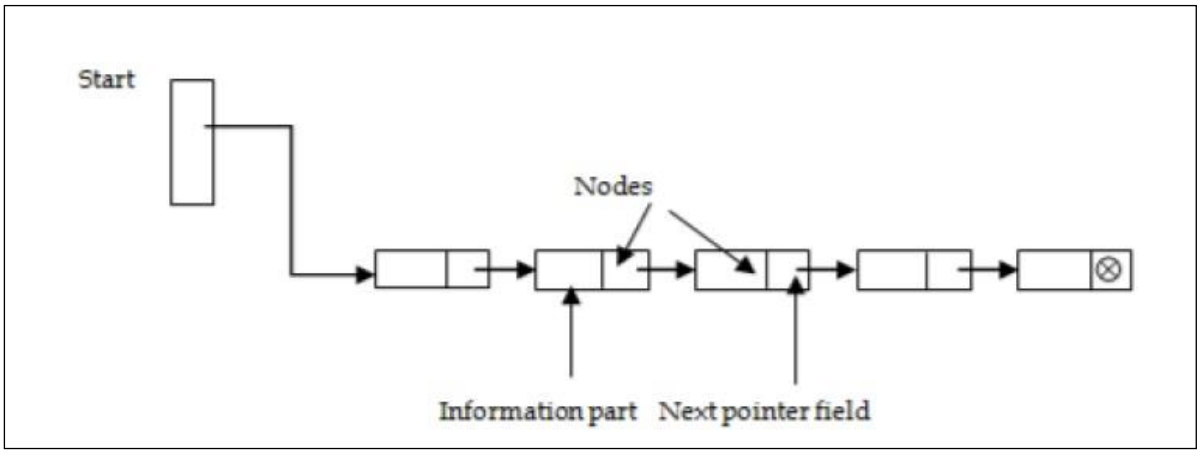


Fig. 2.3: A Linked List

Advantage: Easy to insert and delete data items.

Disadvantage: Searching a data item requires extra memory space and is slow.

Applications

1) Implement stacks, queues, binary trees, and graphs of predefined size.
2) Implement dynamic memory management functions of the operating system (OS).
3) Circular linked list can implement OS or application functions for round robin execution of tasks.
4) Doubly linked list is used in the implementation of forward and backward buttons of a browser.

Stack

A stack is a linear data structure in which insertion and deletion occur at the top of the stack. It is called a last-in first-out (LIFO) data structure because the last element that is pushed on to the top of the stack is always the first one to be popped off or deleted.

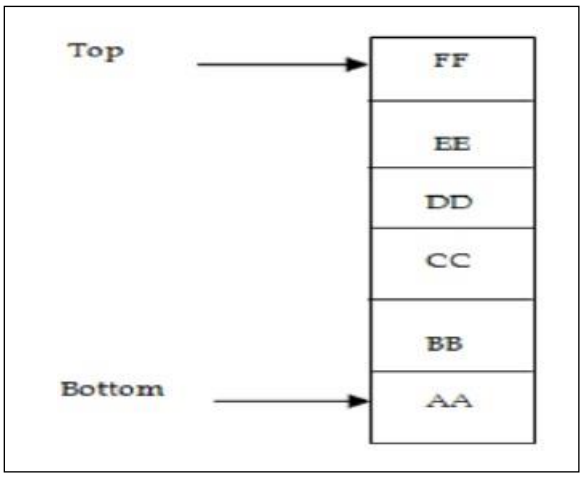


Fig. 2.4: A Stack

In the computer's memory, stacks can be implemented using arrays or linked lists. Fig. 2.4 shows the schematic diagram of a stack. FF is the top of the stack and AA is the bottom of the stack. Since the stack is implemented in a LIFO pattern, data element EE cannot be popped off or deleted before FF. Similarly, DD cannot be deleted before EE.

Applications:

1) Temporary storage structure in recursive operation.
2) Auxiliary storage structure for nested operations and function calls.
3) Management of function calls.
4) Evaluation of arithmetic expressions in various programming languages
5) Checking of syntax expressions
6) Matching of parenthesis
7) String reversal
8) Solutions to problems with regards to backtracking
9) Conversion of infix expressions to postfix expressions
10) Used in depth-first search in graphs and tree traversal
11) Operating system functions
12) Used in undo and redo operations of an editor

Queues

A queue is a first-in and first-out data structure so that the first element that is inserted into it is the first one to be removed. Data elements are added at the rear of the queue and removed from the front of the queue. Like stacks, queues can be implemented using arrays or linked list.

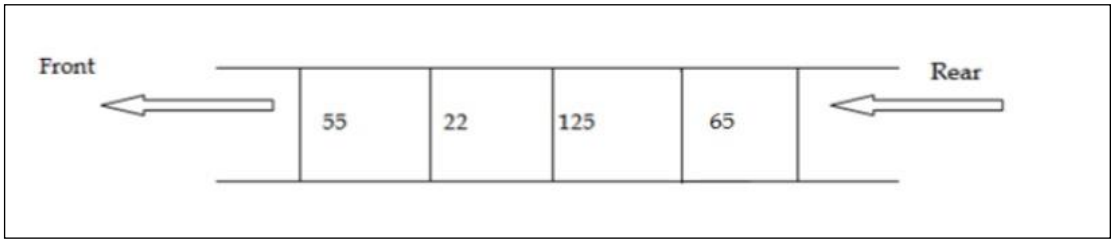


Fig. 2.5: A Queue

Figure 2.5 shows a queue with 4 elements, where 55 is the front element and 65 is the rear element. Elements can be added from the rear and deleted from the front.

Applications:

1) It is used in breadth-first search in graphs
2) Job scheduler operations of OS like a print buffer queue and keyboard buffer queue to store the keys pressed by users.
3) Job scheduling, CPU scheduling, and Disk Scheduling
4) Priority queues are used in file downloading operations in a browser
5) Data transfer between peripheral devices and CPU
6) Interrupts generated by the user applications for CPU

2.8 Operations on Data Structures

Here, we discuss the various data operations that can be performed on the data structures previously mentioned.

Traversing: Here we access every data item of the data structure only once so that it can be processed. For example, we print the names of students in a class.

Searching: We find the location(s) of one or more data items in a data collection, which satisfy the given condition. Such a data item may not be present in the collection. For example, we try to find the names of students who secured a 100 in Mathematics.

Inserting: We add new item(s) to the existing list of data items. For example, we add the details of new students who recently enrolled in a course.

Deleting: We remove (delete) data item(s) from a list of data items. For example, we delete student name(s) who have abandoned a course.

Sorting: We arrange the data items of a data collection in ascending or descending order. For example, we arrange the student names in a class in alphabetical order.

Merging: We combine two separate sorted lists of data items into one sorted list of data items.

2.9 Abstract Data Type

The operations in data structure involve higher-level abstractions such as, adding or deleting an item from a list, accessing the highest priority item in a list, or searching and sorting an item in a list. When the data structure does such operations, it is called an abstract data type.

It can be defined as a collection of data items together with the operations on the data. The word "abstract" refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves what can be done with the data, not how it has to be done.

2.10 Advantage using Abstract Data Trees

In the real world, programs evolve as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data

structures. For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency.

In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to separate the use of a data structure from the details of its implementation. This is where abstract data types come into play and can be helpful.

An implementation of Abstract Data Trees (ADT) consists of storage structures to store the data items and algorithms for basic operation. All the data structures i.e., array, linked list, stack, queue etc. are examples of ADT.

C H A P T E R 3

Why Algorithms?

3. Defining an Algorithm

An algorithm is a step-by-step procedure which contains a set of instructions in a particular order to get a desired output. An algorithm is independent of the underlying programming languages.

From data structure point of view, an algorithm is used to

1) **Search**- for searching a data item within a data structure.
2) **Sort**- for sorting data items in a certain order.
3) **Insert**- for inserting a data item in a data structure.
4) **Update** - for updating an existing data item in a data structure
5) **Delete** - for deleting an existing data item in a data structure

3.1 Features of an Algorithm

Not all procedures can be called an algorithm. An algorithm has the following features:

1) Clear and Unambiguous – Every step in the algorithm should be clear in all angles and should lead to one meaning.
2) Well-Defined Inputs – If an algorithm asks for inputs, they should be well-defined inputs.
3) Well-Defined Outputs – An algorithm should also clearly define its output.
4) Finiteness – The algorithm should not end up in infinite loops; rather they should be clearly finite.
5) Feasible – The algorithm should be simple, generic and practical and should be able to be executed upon available resources. It should not redundantly contain some future technology.
6) Language-Independent – An algorithm should be clearly independent of any language and contain instructions that can be implemented in any language. And the output will be the same, as expected.

3.2 Advantages and Disadvantages of an Algorithm

Advantages

1) It is easy to comprehend.
2) An algorithm is the step-by-step solution to a specific problem.
3) In an algorithm, the problem is broken down into smaller steps so that it is easier for the programmer to convert it to a program.

Disadvantages

1) Writing an algorithm takes a long time so that time is wasted.
2) Branching and looping are difficult to incorporate in an algorithm.

3.3 Different Approaches to Designing an Algorithm

1) Top-Down Approach

This approach starts with identifying major system components and splitting them down into lower-level components and iterating them until a desired level of complexity is attained. In this approach, we begin with the top-most module and incrementally, add lower-level modules.

2) Bottom-Up Approach

This approach starts with designing the basic or primitive component and forms higher level components. Starting from the bottom, operations that cater to abstract layering are implemented.

3.4 How to Write an Algorithm

There are no well-defined standards for algorithms. Rather it is problem and resource dependent. Algorithms are never written to support a particular programming language. We write algorithms in a step-by-step manner, but that is not always the case. Writing an algorithm is a process, which is executed after the problem is well-defined.

Example Problem: Design an algorithm to add two numbers and display the result.

Step 1: Start

Step 2: declare three integers a, b & c

Step 3: define values of a & b

Step 4: add values of a & b

Step 5: store output of Step 4 in c

Step 6: print c

Step 7: Stop

Algorithms tell programmers how to code the program. Alternatively, the above program can be written as below.

Step 1: Start ADD

Step 2: Add values of a & b

Step 3: c ← a + b

Step 4: display c

Step 5: Stop

In design and analysis of algorithms, the second method is preferred because it makes it easy for the analyst to observe what operations are being used and how the process is flowing.

Writing step numbers is optional. An algorithm can be designed in more than one way to give the solution to a problem. The next step is to consider the proposed solutions of algorithms and choose the best suitable solution.

3.5 Algorithmic Complexity

Suppose A is an algorithm and n is the size of input data. The time and space of the algorithm A are the two main factors, which define the efficiency of A.

1) **The Time Factor:** Time is measured by counting the number of key operations such as, the number of comparisons in a sorting algorithm.
2) **The Space Factor:** Space is measured by counting the maximum number of memory locations required by the algorithm.

The algorithmic complexity is given by the function f(n) based on the running time and/or the storage space required by the algorithm in terms of n, the size of input data.

3.6 Space Complexity

Space complexity of an algorithm is the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is the sum of two components:

1) The fixed part is the space required by certain data and variables, independent of the size of the problem. Examples are simple variables and constants, program size.
2) The variable part is the space required by variables, dependent on the size of the problem. Examples are dynamic memory allocations and recursion stack space.

The space complexity of an algorithm P can, therefore, be expressed as S(P)= C + S(I) where C is the fixed part and S(I) is the variable part, dependent on the instance characteristic I.

The following is a simple example of the above:

Algorithm SUM(A,B)

Step 1: Start

Step 2: C ← A + B + 10

Step 3: Stop

Here we have three variables A, B and C and one constant, 10. So, S(P)= 1 + 3. Now, space depends on the data types of the variables and the constant type, and they will be multiplied accordingly.

3.7 Time Complexity

Time complexity of an algorithm is the time required by the algorithm to run to completion. Time measurement can be expressed as a function T(n) representing the time taken to complete n steps, provided each step requires constant time.

For example, addition of 2 n-bit integers takes n steps. Then the computational time T(n) = c * n where the c is the time taken to add 2 n-bits. Here, we observe T(n) grows linearly as the input size increases.

3.8 Analysis of Algorithms

Efficiency of an algorithm can be computed before or after the implementation stages of the algorithm. They can be categorized as follows:

1) **A Priori Analysis or Performance/Asymptotic Analysis**: This is the theoretical analysis that all other factors including the processor speed are constant and have no effect on implementation.
2) **A Posterior Analysis or Performance Measurement**: This is the empirical analysis of an algorithm. The algorithm is implemented using a programming language and executed on a target computer machine. The statistics such as the running time and required space are collected. Also, the input size is considered.

In theoretical analysis, the complexity of the algorithm is estimated in the asymptotic sense for arbitrarily large input. Big-O notation, Omega notation and Theta notation (explained later) are used to estimate the complexity function of the algorithm for arbitrary large input.

Types of Analysis

The efficiency of some algorithms may vary owing to inputs of the same size. These are the best, average and worst-case efficiencies.

The Best-Case Analysis

If an algorithm takes the least amount of time to execute a set of inputs of the same size, then it is a best-case complexity. The best-case efficiency of the algorithm is the efficiency for the best-case input of size n. The algorithm runs the fastest for all the possible inputs of the same size.

The Average Case Analysis

If the time taken by an algorithm for some sets of inputs is on average, then the time complexity of the algorithm is the average case time complexity.

You must make some assumption about the possible inputs of size n to compute the average case time complexity of the algorithm.

The Worst-Case Analysis

If an algorithm takes the maximum time to compute for a set of inputs of the same size, then it is a worst-case time complexity. The worst-case efficiency is the efficiency for the worst case of inputs of size n. The algorithm runs the longest for all the possible inputs of similar size n.

3.9 Mathematical Notations

Many problems can be solved using the same algorithm. Therefore, the algorithms must follow a standard. Mathematical notations use symbols or symbolic expressions to give semantic meaning.

Asymptotic Notations

An algorithm can have many solutions. The efficiency of a solution can be found by computing the time complexity of the algorithm, and the best algorithm can be sought. The asymptotic notations help to represent the time complexity of the algorithm. It can be represented as best possible, worst possible or average possible.

The notations such as O (Big-O), Ω (Omega), and θ (Theta) are three asymptotic notations to represent three different cases of time complexity for the algorithm.

Big-O Notation

O is the notation for Big-O notation. It represents upper bound of the running time of the algorithm and describes the worst-case scenario for the time execution or space

required by the algorithm. This, in fact, describes the time complexity or performance of the algorithm.

Big-O notation is used represent the maximum time required to run an algorithm. Big-O is defined as:

f(n) <= n * g(n)

where n can be any number of inputs or outputs and f(g) and g(n) are two non-negative functions. These functions are only true if there is a constant c and a nonnegative integer n0 such that n >= n0.

The Big-O notation is expressed as f(n)= O(g(n)), where f(n) and g(n) are two nonnegative functions such that f(n) < g(n) such that g(n) is a multiple of some constant c. The graphical representation of f(n) = O(g(n)) is shown in Fig. 3.1, where the running time increases considerably when n increases**.**

Table 3.1: Common Orders

Time Complexity		**Examples**
O(1)	Constant	Constant Adding to the front of a linked list
O(log n)	Logarithmic	Finding an entry in a sorted array
O (n)	Linear	Finding an entry in an unsorted array
O(n log n)	Linearithmic	Sorting 'n' items by 'divide-and-conquer'
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Simultaneous linear equations
$O(2^n)$	Exponential	The Towers of Hanoi problem

Example: Consider $f(n)=15n^3+40n^2+2n\log n+2n$. As the value of n increases, n^3 becomes much larger in value than n^2, n log n, and n. Hence, it dominates the function f(n), and we can consider the running time to grow by the order of n^3. Therefore, it can be written as $f(n)=O(n^3)$.

The values of n for f(n) and c * g(n) will not be less than n0. Therefore, the values less than n0 are not considered relevant.

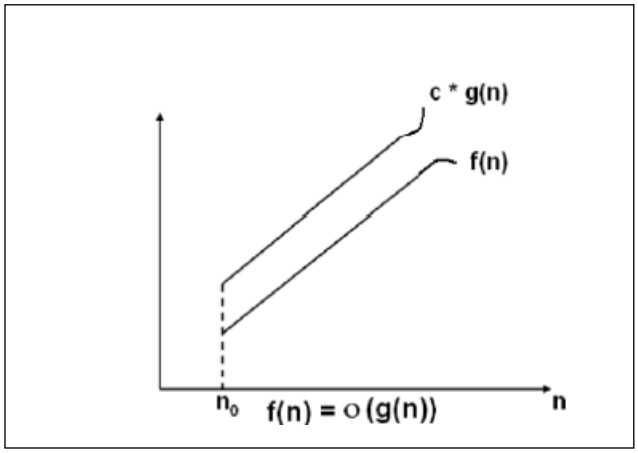


Fig. 3.1: Big-O Notation, f(n)=O(g(n))

Let's try to understand Big-O notation more lucidly.

Example:
Consider function f(n) = 2(n)+2 and g(n) = n^2.
We need to find n such that f(n) ≤ c ∗ g(n). Here c=1 (co-efficient of n^2 in function g(n).)

Let n = 1, then
f(n) = 2(n)+2 = 2(1)+2 = 4
g(n) = n^2 = 1^2 = 1
Here, f(n)>g(n)

Let n = 2, then
f(n) = 2(n)+2 = 2(2)+2 = 6
g(n) = n^2 = 2^2 = 4
Here, f(n)>g(n)

Let n = 3, then
f(n) = 2(n)+2 = 2(3)+2 = 8
g(n) = n^2 = 3^2 = 9
Here, f(n)<g(n)

Thus, when n is greater than 2, we get f(n)<g(n). In other words, as n becomes larger, the running time increases considerably. This concludes that the Big-O helps to determine the 'upper bound' of the algorithm's run-time.

Limitations of Big-O notation

Big-O notations have certain limitations. These are explained as below:

1) Many algorithms are hard to analyze mathematically using Big-O notation.
2) We may not have sufficient info to compute the complexity of the algorithm in the average case.
3) Big-O analysis does not specify the efficiency of the algorithm as it grows with the size of the problem and doesn't take the programming aspect into effect.
4) It ignores constants. If a program takes $O(n^2)$ time and another one takes $O(1000n^2)$ time, then according to Big-O analysis, the constant is ignored and both would require $O(n^2)$ time complexity. In real-time systems, this may be a serious issue.

Omega Notation

'Ω' represents Omega notation. Omega describes the manner in which an algorithm performs the best-case time complexity. This notation describes, therefore, the minimum running time of the algorithm. In fact, it gives the lower bound on the running time of the algorithm. Omega is defined as:

$$f(n) >= c * g(n)$$

where n is any number of inputs or outputs and f(n) and g(n) are non-negative functions. The functions are only true if there is a constant c and n0 is a nonnegative integer such that n>n0.

Omega can also be denoted as f of n is equal to omega of g of n.

$$f(n) = \Omega(g(n))$$

Omega notation is shown graphically as below:

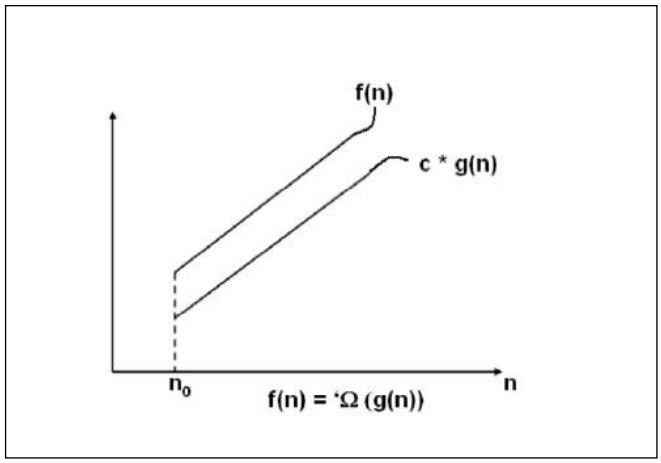


Fig. 3.2: Omega Notation, f(n) = Ω(g(n))

Consider function $f(n) = 2n^2+5$ and $g(n) = 7n$.
We need to find n such that $f(n) \geq c * g(n)$.

Let n = 0, then
$f(n) = 2n^2+5 = 2(0)^2+5 = 5$
$g(n) = 7(n) = 7(0) = 0$
Here, $f(n)>g(n)$

Let n = 1, then
$f(n) = 2n^2+5 = 2(1)^2+5 = 7$
$g(n) = 7(n) = 7(1) = 7$
Here, $f(n)=g(n)$

Let n = 2, then
$f(n) = 2n^2+5 = 2(2)^2+5 = 13$
$g(n) = 7(n) = 7(2) = 14$
Here, $f(n)<g(n)$

Thus, for n = 1, we got $f(n) = c * g(n)$. Let us take another example:
Consider function $f(n)=2(n)+2$ and $g(n) = n^2$.

Let n=1, then
$f(n) = 2(n)+2 = 2(1)+2 = 4$
$g(n) = n^2 = 1^2 = 1$
Here, $f(n)>g(n)$

Thus, for n=1, we get $f(n) \geq c * g(n)$.

This concludes that Omega helps to determine the "lower bound" of the algorithm's run-time.

Theta Notation

'θ' represents Theta Notation. Theta notation is used when the upper bound and lower bound of an algorithm are in the same order of magnitude. Theta can be defined as:

$c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n>n0$

where, n is any number of inputs or outputs and f(n) and g(n) are two nonnegative functions. These functions are true only if there are two constants namely, c1, c2**,** and a non-negative integer n0.

Theta can also be denoted as f(n) = θ(g(n)) where, f of n is equal to Theta of g of n. The graphical representation of f(n) = θ(g(n)) is shown in Fig. 3.4. The function f(n) is said to be in θ(g(n)) if f(n) is bounded both above and below by some positive constant multiples of g(n) for all large values of n.

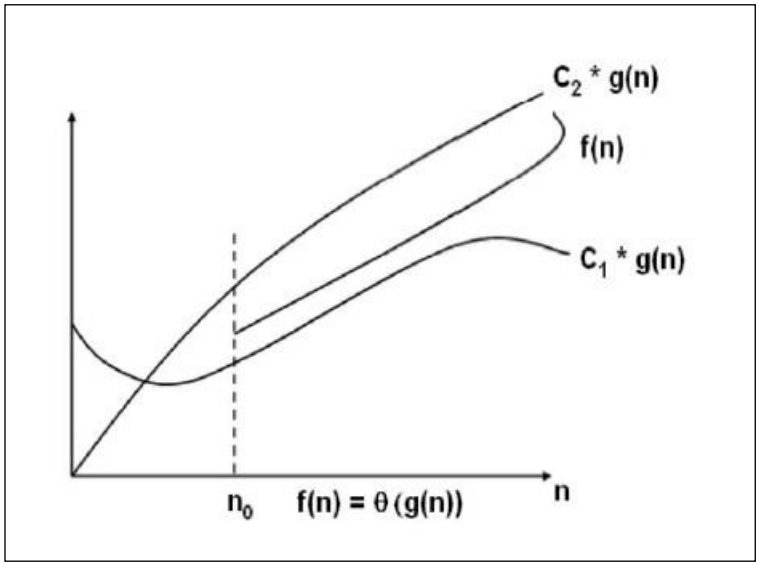


Fig. 3.3: Theta Notation**,** f(n) = θ(g(n))

Example: Consider function f(n) = 4n + 3 and g(n) = 4n for all n ≥ 3; and f(n) = 4n + 3 and g(n) = 5n for all n ≥ 3.

Then the result of the function will be:
Let n = 3
f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 4n =4(3) = 12 and

f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 5n =5(3) = 15 and

here, c1 is 4, c2 is 5 and n0 is 3.

Thus, from the above equation we get c1 g(n)<= f(n)<=c2 g(n). This concludes that Theta notation depicts the running time between the upper bound and lower bound.

C H A P T E R 4

Searching

4. Introduction to Searching Algorithm

Arrays are one of the simplest possible ways of representing collections of numbers (or strings, or whatever), so we shall use that to store the information to be searched. Later we shall look at more complex data structures that may make storing and searching more efficient.

Suppose, for example, that the set of integers we wish to search is 1,4,17,3,90,79,4,6,81.

We can write them in an array a as
a = [1, 4, 17, 3, 90, 79, 4, 6, 81]

If we ask where 17 is in this array, the answer is 2, the index of that element. If we ask where 91 is, the answer is nowhere. It is useful to be able to represent nowhere by a number that is not used as a possible index. Since we start our index counting from 0, any negative number would do. We shall follow the convention of using the number -1 to represent nowhere. Better conventions are possible, but we will stick to this here.

4.1 Specification of the Search Problem

We can now formulate a specification of our search problem using the data structure:

Given an array a and integer x, find an integer i such that:

1. if there is no j such that a[j] is x, then i is -1,

2. otherwise, i is any j for which a[j] is x.

The first condition says that if x does not occur in the array a then i should be -1, and the second says that if it does occur then i should be a position where it occurs. If there is more than one position where x occurs, then this specification allows you to return any one of them. For example, this would be the case if 'a' were [17, 13, 17] and x were 17. Thus, the specification is ambiguous. Hence different algorithms with different behaviors can satisfy the same specification. For example, one algorithm may return the

smallest position at which x occurs, and another may return the largest. There is nothing wrong with ambiguous specifications.

4.2 A Simple Algorithm on Linear Search

Here is a simple algorithm on Linear Search.

```
// This assumes we are given an array a of size n and a key x.
For i = 0,1,...,n-1,
        if a[i] is equal to x,
                then we have a suitable i and can terminate returning i.
If we reach this point,
        then x is not in a and hence we must terminate returning -1.
```

In a programming language such as C, one would write something that is more precise like:

```
for ( i = 0 ; i < n ; i++ ) {
   if ( a[i] == x ) return i;
}
return -1;
```

It is easy to see that the algorithm satisfies the specification (assuming n is the correct size of the array). We start counting from zero, and the last position of the array is its size minus one. If we forget this, and let i run from 0 to n instead, we get an incorrect algorithm.

The practical effect of this mistake is that the execution of this algorithm gives rise to an error when the item to be located in the array is actually not there, because a non-existing location is attempted to be accessed.

4.3 A More Efficient algorithm: Binary Search

One always needs to consider whether it is possible to improve upon the performance of a particular algorithm, such as the one we have just created. In the worst case, searching an array of size n takes n steps. On average, it will take n/2 steps. For large collections of data, such as all web-pages on the internet, this will be unacceptable in practice. Thus, we should try to organize the collection in such a way that a more efficient algorithm is possible.

Here we consider a simple but more efficient algorithm. We still represent the collections by arrays, but now we sort the elements in ascending order.

Thus, instead of working with the previous array [1, 4, 17, 3, 90, 79, 4, 6, 81], we would work with [1, 3, 4, 4, 6, 17, 79, 81, 90], which has the same items but listed in ascending order. Then we can use an improved algorithm, which is as follows:

// This assumes we are given a sorted array a of size n and a key x.
// Use integers left and right (initially set to 0 and n-1) and mid.

While left is less than right,
    set mid to the integer part of (left+right)/2, and
    if x is greater than a[mid],
        then set left to mid+1,
        otherwise set right to mid.
If a[left] is equal to x,
    then terminate returning left,
    otherwise terminate returning -1.

and would correspond to a segment of C code like:

```
/* DATA */
int a = [1,3,4,4,6,17,79,81,90];
int n = 9;
int x = 79;
/* PROGRAM */
int left = 0, right = n-1, mid;
while ( left < right ) {
    mid = ( left + right ) / 2;
    if ( x > a[mid] ) left = mid+1;
    else right = mid;
}
if ( a[left] == x ) return left;
else return -1;
```

This algorithm works by repeatedly splitting the array into two segments, one going from left to mid, and the other going from mid + 1 to right, where mid is the position half way from left to right, and where, initially, left and right are the leftmost and rightmost positions of the array.

Because the array is sorted, it is easy to see which of each pair of segments the searched-for item x is in, and the search can then be restricted to that segment. Moreover, because the size of the sub-array going from locations left to right is halved at each iteration of the while-loop, we only need $\log_2 n$ steps in either the average or worst case. To see that this runtime behavior is a big improvement, in practice, over the earlier linear-search algorithm, notice that $\log_2 1000000$ is approximately 20, so that for an array of size 1000000 only 20 iterations are needed in the worst case of the binary-search algorithm, whereas 1000000 are needed in the worst case of the linear-search algorithm.

How to Calculate Time Complexity

Let's say the iteration in Binary Search terminates after **k** iterations.
At each iteration, the array is divided by half. So, let's say the length of array at any iteration is **N**.

At **Iteration 1,**
Length of array = N

At **Iteration 2**,
Length of array = N/2

At **Iteration 3**,
Length of array = (N/2)/2 = N/2^2

At **Iteration k**,
Length of array = $(N/2)/2^{k-1} = N/(2.2^k.2^{-1}) = N/2^k$

Also, we know that after k divisions, the length of the array becomes 1: Length of array = $\mathbf{N/2^k = 1 \Rightarrow N = 2^k}$

If we apply a log function on both sides: $\mathbf{\log_2 (N) = \log_2 (2^k) \Rightarrow \log_2 (N) = k \log_2 (2)}$

As $\mathbf{(\log_a(a) = 1)}$

Therefore, => $\mathbf{k = \log_2 (N)}$
So now we can see why the time complexity of Binary Search is $\log_2 (N)$.

C H A P T E R 5

Trees

5. Introduction to Trees

In computer science, a tree is a very general and powerful data structure that resembles a real tree. It consists of an ordered set of linked nodes in a connected graph, in which each node has at most one parent node, and zero or more child nodes in a specific order.

5.1 Specification of Trees

Generally, we can specify a tree as consisting of nodes (also called vertices) and edges (also called connecting lines) with a tree-like structure. It is usually easiest to represent trees pictorially, so we shall frequently do that. A simple example is given in Fig. 5.1.

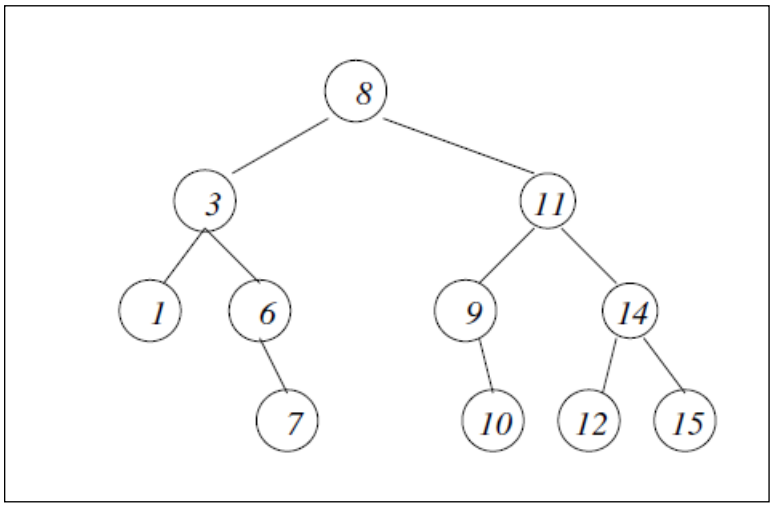


Fig. 5.1: An Example of a Tree

A tree can be defined formally as either the empty tree, or a node with a list of successive trees. Nodes are usually labelled with a data item (such as a number or search key). We will refer to the label of a node as its value. In our examples, we will generally use nodes labelled by integers, but we could easily choose something else, e.g., strings of characters.

There always has to be a unique 'top level' node known as the root. In Figure 5.1, this is the node labelled with 8. It is important to note that, in computer science, trees are normally displayed upside-down, with the root forming the top level. Then, given a node, every node on the next level down that is connected to the given node via an edge, is a child of that node.

In Fig. 5.1, the children of node 8 are nodes 3 and 11. Conversely, the node connected to the given child node (via an edge) on the level above, is its parent. For instance, node 11 is the parent of node 9 and of node 14 as well. Nodes that have the same parent are known as siblings; siblings are, by definition, always on the same level.

If a node is the child of a child of . . . of another node, then we say that the first node is a descendent of the second node. Conversely, the second node is an ancestor of the first node. Nodes which do not have any children are known as leaves (e.g., the nodes labelled with 1, 7,10, 12, and 15 in Fig. 5.1).

A path is a sequence of connected edges from one node to another. Trees have the property that for every node there is a unique path connecting it with the root. In fact, this is another possible definition of a tree. The depth or level of a node is given by the length of this path. Therefore, the root has level 0, its children have level 1, and so on.

The maximal length of a path in a tree is also called the height of the tree. A path of maximal length always goes from the root to a leaf. The size of a tree is given by the number of nodes it contains. We shall normally assume that every tree is finite. The tree in Fig. 5.1 has height 3 and size 11.

A tree consisting of just of one node has height 0 and size 1. The empty tree obviously has size 0 and is defined somewhat to have height -1.

Like most data structures, we need a set of primitive operators (constructors, selectors and conditions) to build and manipulate the trees. The details of these depend on the type and purpose of the tree. We will now look at some particularly useful types of trees.

5.2 Quadtrees

A quadtree is a particular type of tree in which each leaf-node is labelled by a value and each non-leaf node has exactly four children. It is used most often to partition a two-dimensional space (e.g., a pixelated image) by recursively dividing it into four quadrants.

Formally, a quadtree can be defined to be either a single node with a number or value (e.g., in the range 0 to 255), or a node without a value but with four quadtree children: lu, ll, ru, and rl. It can thus be defined "inductively" by the following rules:

Definition: A quadtree is either
(Rule 1) a root node with a value, or
(Rule 2) a root node without a value and four quad tree children: lu, ll, ru, and rl.

in which Rule 1 is the base case and Rule 2 is the induction step.

What do we mean by base case and induction step? Here goes:

The base case (or initial case): prove that the statement holds for 0, or 1. The induction step (or inductive step, or step case): prove that for every n, if the statement holds for n, then it holds for n + 1.

We say that a quadtree is primitive if it consists of a single node/number, and that can be tested by the corresponding condition:

isValue(qt), which returns true if quad-tree qt is a single node.

To build a quadtree we have two constructors:

baseQT(value), which returns a single node quad-tree with label value.

makeQT(luqt, ruqt, llqt, rlqt), which builds a quad-tree from four constituent quad-trees luqt, llqt, ruqt, rlqt.

Then to extract components from a quad-tree we have four selectors:

lu(qt), which returns the left-upper quad-tree.

ru(qt), which returns the right-upper quad-tree.

ll(qt), which returns the left-lower quad-tree.

rl(qt), which returns the right-lower quad-tree.

- which can be applied whenever isValue(qt) is false.

Quad-trees of this type are most commonly used to store grey-value pictures (with 0 representing black and 255 white). A simple example would be as shown below in Fig.5.2:

We can then create algorithms using the operators to perform useful manipulations of the representation. For example, we could rotate a picture qt by 180 using:

```
rotate(qt) {
if ( isValue(qt) )
   return qt
else return makeQT(rotate(rl(qt)), rotate(ll(qt)), rotate(ru(qt)), rotate(lu(qt))
}
```

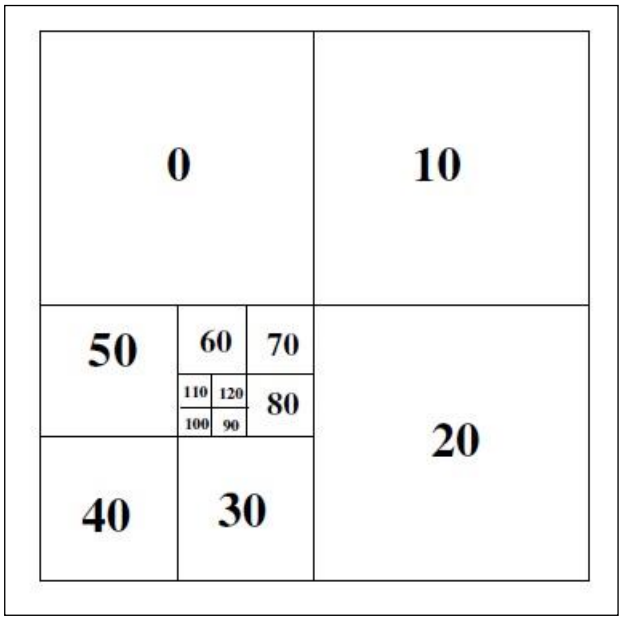


Fig. 5.2: A Quadtree Example

5.3 Binary Trees

Binary trees are the most common type of tree used in computer science. A binary tree is a tree in which every node has at most two children, and can be designed inductively by the following rules:

Definition: A binary tree is either

(Rule 1) the empty tree EmptyTree, or

(Rule 2) it consists of a node and two binary trees, the left subtree and right subtree.

Again, Rule 1 is the base case and Rule 2 is the induction step.

You can imagine that the (infinite) collection of (finite) trees is created in a sequence of days. Day 0 is when you get off the ground by applying Rule 1 to get the empty tree. On later days, you are allowed to use any trees that you have created on earlier days to construct new trees using Rule 2. Thus, for example, on day 1 you can create exactly trees that have a root with a value, but no children (i.e., both the left and right subtrees are the empty trees, created at day 0).

On day 2 you can use a new node with value, with the empty tree and/or the one-node tree, to create more trees. Thus, binary trees are the objects created by the above two rules in a finite number of steps. The height of a tree, defined above, is the number of days it takes to create it using the above two rules, where we assume that only one rule is used per day, as we have just discussed.

5.3.1 Primitive Operations on Binary Trees

The primitive operators for binary trees are fairly obvious. We have two constructors which are used to build trees:

- EmptyTree, which returns an empty tree,

- MakeTree(v, l, r), which builds a binary tree from a root node with label v and two constituent binary trees l and r.

A condition to test whether a tree is empty:
- isEmpty(t), which returns true if tree t is the EmptyTree.

Three selectors to break a non-empty tree into its constituent parts:

- root(t), which returns the value of the root node of binary tree t,

- left(t), which returns the left sub-tree of binary tree t,

- right(t), which returns the right sub-tree of binary tree t.

These operators can be used to create all the algorithms we might need for manipulating binary trees. For convenience though, it is often a good idea to define derived operators that allow us to write simpler, more readable algorithms. For example, we can define a derived constructor:

Leaf(v) = MakeTree(v, EmptyTree, EmptyTree)

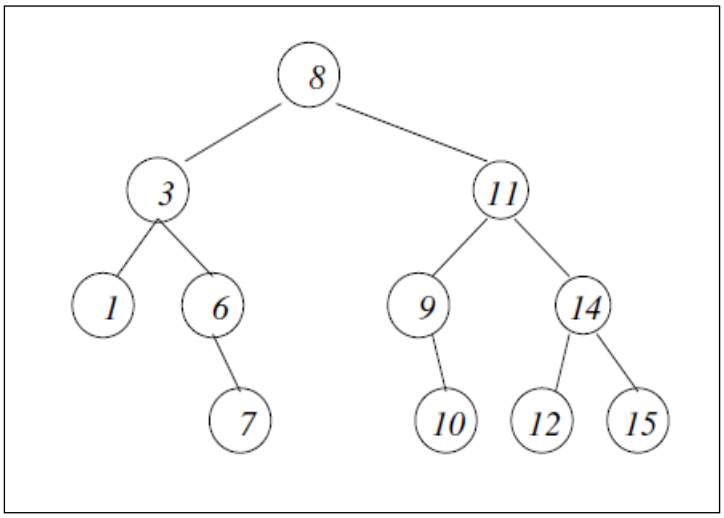
that creates a tree consisting of a single node with label v, which is the root and the unique leaf of the tree at the same time. Then the tree in Fig. 5.1 can be constructed as:

t = MakeTree(8, MakeTree(3, Leaf(1), MakeTree(6, EmptyTree, Leaf(7))),
MakeTree(11, MakeTree(9, EmptyTree, Leaf(10)), MakeTree(14, Leaf(12), Leaf(15))))

which is much simpler than the construction using the primitive operators:
t = MakeTree(8, MakeTree(3,MakeTree(1,EmptyTree,EmptyTree),
MakeTree(6,EmptyTree,MakeTree(7,EmptyTree,EmptyTree))),

MakeTree(11,MakeTree(9,EmptyTree,MakeTree(10,EmptyTree,EmptyTree)),
MakeTree(14,MakeTree(12,EmptyTree,EmptyTree),
MakeTree(15,EmptyTree,EmptyTree)))

Here is Fig. 5.1 again for reference:



Note that the selectors can only operate on non-empty trees. For example, for the tree t, defined above we have:

root(left(left(t)) = 1,

but the expression

root(left(left(left(t)))) does not make sense because

left(left(left(t))) = EmptyTree

and the empty tree does not have a root. In a language such as C, this would cause an unpredictable behavior, and the program may be aborted with no further harm. When writing algorithms, we need to check the selector arguments using:

isEmpty(t) before allowing their use.

The following equations should be obvious from the primitive operator definitions:

root(MakeTree(v,l,r)) = v

left(MakeTree(v,l,r)) = l

right(MakeTree(v,l,r)) = r

isEmpty(EmptyTree) = true

isEmpty(MakeTree(v,l,r)) = false

The following makes sense only under the assumption that t is a non-empty tree:

MakeTree(root(t), left(t), right(t)) = t

It just says that if we break apart a non-empty tree and use the pieces to build a new tree, then we get an identical tree back. It is worth emphasizing that the above specifications of quadtrees and binary trees are further examples of abstract data types: data types for which we exhibit the constructors and destructors and describe their behavior (using equations which we have defined earlier for lists, stacks, queues, quadtrees and binary trees), but for which we explicitly hide the implementational details.

5.3.2 The Height of a Binary Tree

A perfectly balanced binary tree is a binary tree in which all internal nodes have exactly two children and all leaves are at the same level.

Let n be the number of nodes in a perfect binary tree and let l_k denote the number of nodes on level k.

Note that:

- $l_k = 2l_{k-1}$, i.e., each level has exactly twice as many nodes as the previous level (since each internal node has exactly two children)

- $l_0 = 1$, i.e., on the "first level" we have only one node (the root node).

- the leaves are at the last level, l_h, where h is the height of the tree. The total number of nodes in the tree is equal to the sum of the nodes on all the levels: nodes n.

Basically, if done appropriately, many important tree-based operations (such as, searching) take as many steps as the height of the tree, so minimizing the height minimizes the time needed to perform those operations.

The total number of nodes in the tree is equal to the sum of the nodes on all the levels: nodes n.

$$1 + 2^1 + 2^2 + 2^3 + ... + 2^h = n$$

In fact, if we call the size function s(h), it seems fairly obvious that

$s(h) = 1 + 2^1 + 2^2 + 2^3 + ... + 2^h = 2^{h+1} - 1.$

This hypothesis can be proved by induction using the definition of a binary tree as follows:

- The base case applies to the empty tree that has height h =-1, which is consistent with $s(-1) = 2^{-1+1} - 1 = 2^0 - 1 = 1 - 1 = 0$ nodes being stored.

- Then for the induction step, a tree of height h + 1 has a root node plus two subtrees of height h. By the induction hypothesis, each subtree can store $2^{h+1} - 1$ nodes; so the total number of nodes that can fit in a height h + 1 tree is

$1 + 2 \times (2^{h+1} - 1) = 1 + 2^{h+2} - 2 = 2^{(h+1)+1} - 1 = s(h + 1).$

It follows that if the base case is correct for the empty tree, then it is correct for all h. Therefore,

$1 + 2^1 + 2^2 + 2^3 + ... + 2^h = n$

$2^{h+1} - 1 = n$

$\log_2(2^{h+1}) = \log_2 (n + 1)$

$(h + 1) \log_2 2 = \log_2 (n + 1)$

$h + 1 = \log_2 (n + 1)$

$h = \log_2 (n + 1) - 1$

Therefore, h is $O(\log_2 n)$

Now that we know the height of the tree, we can compute the number of leaves, l_h, in the tree. We observed earlier that $l_h = 2^h$; so, we can substitute the value of h in this expression as shown below:

$2^h = 2^{\log_2(n+1)-1} = 2^{\log_2(n+1)}/2^1 = (n + 1)/2$

The number of leaves is $l^h = (n + 1)/2$, i.e. roughly half of the nodes are at the leaves.

5.3.3 The Size of a Binary Tree

Usually, a binary tree will not be perfectly balanced, so we will need an algorithm to determine its size, i.e., the number of nodes it contains.

This is easy if we use recursion. The terminating case is very simple: the empty tree has size 0. Otherwise, any binary tree will always be assembled from a root node, a left sub-tree l, and a right sub-tree r, and its size will be the sum of the sizes of its components, i.e., 1 for the root, plus the size of l, plus the size of r.

We have already defined the primitive operator isEmpty(t) to check whether a binary tree t is empty, and the selectors left(t) and right(t) which return the left and right sub-trees of binary tree t. Thus, we can easily define the procedure size(t), which takes a binary tree t and returns its size, as follows:

```
size(t) {
  if ( isEmpty(t) )
      return 0
   else return (1 + size(left(t)) + size(right(t)))
}
```

This recursively processes the whole tree, and we know it will terminate because the trees being processed get smaller with each call, and will eventually reach an empty tree which returns a zero value.

C H A P T E R 6

Binary Search Tree

6. Definition

The data to be searched needs to be stored using a binary tree in such a way that searching for a particular item takes minimal effort. The underlying idea is simple: At each tree node, we want the value of that node to either tell us that we have found the required item, or tell us in which of its two subtrees we should search for it. For the moment, we shall assume that all the items in the data collection are distinct, with different search keys, so each possible node value occurs at most once.

Definition. A binary search tree is a binary tree that is either empty or satisfies the following conditions:

- All values occurring in the left subtree are smaller than that of the root.
- All values occurring in the right subtree are larger than that of the root.
- The left and right subtrees are themselves binary search trees.

So, this is just a particular type of binary tree, with node values that are the search keys. This means we can inherit many of the operators and algorithms we defined for general binary trees. In particular, the primitive operators MakeTree(v, l, r), root(t), left(t), right(t), and isEmpty(t) are the same- we just need to maintain the additional node value ordering.

6.1 Building Binary Search Trees

When building a binary search tree, one naturally starts with the root and then adds further new nodes as needed. So, to insert a new value v, the following cases arise:

- If the given tree is empty, then simply assign the new value v to the root, and leave the left and right subtrees empty.

- If the given tree is non-empty, then insert a node with value v as follows:

- If v is smaller than the value of the root: insert v into the left sub-tree.
- If v is larger than the value of the root: insert v into the right sub-tree.
- If v is equal to the value of the root: report a violated assumption.

Thus, using the primitive binary tree operators, we have the procedure:

```
insert(v,bst) {
if ( isEmpty(bst) )
   return MakeTree(v, EmptyTree, EmptyTree)
elseif ( v < root(bst) )
   return MakeTree(root(bst), insert(v,left(bst)), right(bst))
elseif ( v > root(bst) )
     return MakeTree(root(bst), left(bst), insert(v,right(bst)))
else error(`Error: violated assumption in procedure insert.')
}
```

which inserts a node with value v into an existing binary search tree bst. Note that the node added is always a leaf. The resulting tree is once again a binary search tree.

6.2 Searching a Binary Search Tree

Searching a binary search tree is not dissimilar to the process performed when inserting a new item. We simply have to compare the item being looked for with the root, and then keep 'pushing' the comparison down into the left or right subtree depending on the result of each root comparison, until a match is found or a leaf is reached.

Algorithms can be expressed in many ways. Here is a concise description in words of the search algorithm that we have just outlined:

In order to search for a value v in a binary search tree t, proceed as follows. If t is empty, then v does not occur in t, and hence we stop with false. Otherwise, if v is equal to the root of t, then v does occur in t, and hence we stop, returning true. If, on the other hand, v is smaller than the root, then, by definition of a binary search tree, it is enough to search the left subtree of t. Hence replace t by its left subtree and carry on in the same way. Similarly, if v is bigger than the root, replace t by its right sub-tree and carry on in the same way.

Notice that such a description of an algorithm covers both the steps that need to be carried out and the reason why this gives a correct solution to the problem. This way of describing algorithms is very common when we do not intend to run them on a computer.

When we do want to run them, we need to provide a more precise specification, and would normally write the algorithm in pseudocode, such as the following recursive procedure:

```
isIn(value v, tree t) {
   if ( isEmpty(t) )
      return false
   elseif ( v == root(t) )
      return true
   elseif ( v < root(t) )
      return isIn(v, left(t))
   else
      return isIn(v, right(t))
}
```

Each recursion restricts the search to either the left or right subtree as appropriate, reducing the search tree height by one, so the algorithm is guaranteed to terminate eventually.

6.3 Time Complexity of Insertion and Search

Carrying out exact tree height calculations is not straightforward, so we will not do that here. However, if we assume that all the possible orders in which a set of n nodes might be inserted into a binary search tree are equally likely, then the average height of a binary search tree turns out to be $O(\log_2 n)$.

It follows that the average number of comparisons needed to search a binary search tree is $O(\log_2 n)$, which is the same complexity we found for binary search of a sorted array. However, inserting a new node into a binary search tree also depends on the tree height and requires $O(\log_2 n)$ steps, which is better than the O(n) complexity of inserting an item into the appropriate point of a sorted array.

6.4 Deleting Nodes from a Binary Search Tree

Suppose, for some reason, an item needs to be removed or deleted from a binary search tree where n items would require n steps of $O(\log_2 n)$ complexity, and hence have overall time complexity of $O(n\log_2 n)$. By comparison, deleting an item from a sorted array would only have time complexity O(n), and we certainly want to do better than that. Instead, we need an algorithm that produces an updated binary search tree more efficiently. This is more complicated than one might assume at first sight, but it turns out that the following algorithm works as desired:

-If the node in question is a leaf, just remove it.

-If only one of the node's subtrees is non-empty,`move up' the remaining subtree.

-If the node has two non-empty subtrees, find the`left-most' node occurring in the right subtree (this is the smallest item in the right subtree). Use this node to overwrite the one that is to be deleted. Replace the left-most node by its right subtree, if this exists; otherwise just delete it.

The last part works because the left-most node in the right sub-tree is guaranteed to be bigger than all nodes in the left subtree, and smaller than all the other nodes in the right subtree, and has no left subtree itself. For instance, if we delete the node with value 11 from the tree in Fig. 5.1, we get the tree displayed in Fig. 6.1.

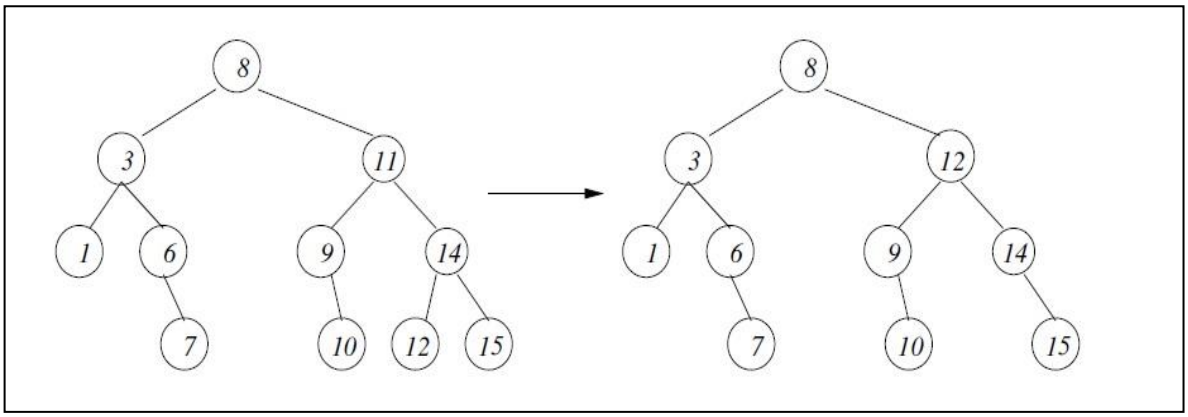


Fig. 6.1: Node Deletion in a Binary Search Tree

In practice, we need to turn the above algorithm (specified in words) into a more detailed algorithm specified using the primitive binary tree operators:

```
delete(value v, tree t) {
    if ( isEmpty(t) )
        error(`Error: given item is not in given tree')
    else
        if ( v < root(t) ) // delete from left sub-tree
            return MakeTree(root(t), delete(v,left(t)), right(t));
    else if ( v > root(t) ) // delete from right sub-tree
            return MakeTree(root(t), left(t), delete(v,right(t)));
     else // the item v to be deleted is root(t)
        if ( isEmpty(left(t)) )
            return right(t)
     elseif ( isEmpty(right(t)) )
        return left(t)
     else // difficult case with both subtrees non-empty
        return MakeTree(smallestNode(right(t)), left(t),
                                removeSmallestNode(right(t))
}
```

If the empty tree condition is met, it means the search item is not in the tree, and an appropriate error message should be returned. The delete procedure uses two sub-algorithms to find and remove the smallest item of a given subtree. Since the relevant subtrees will always be non-empty, these sub-algorithms can be written with that precondition.

First, to find the smallest node, we have:

```
 smallestNode(tree t) {
    // Precondition: t is a non-empty binary search tree
 if ( isEmpty(left(t) )
     return root(t)
 else
      return smallestNode(left(t));
 }
```

which uses the fact that, by the definition of a binary search tree, the smallest node of t is the left-most node. It recursively looks in the left sub-tree till it reaches an empty tree, at which point it can return the root. The second sub-algorithm uses the same idea:

```
 removeSmallestNode(tree t) {
    // Precondition: t is a non-empty binary search tree
 if ( isEmpty(left(t)
    return right(t)
 else
     return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

except that the remaining tree is returned rather than the smallest node. These procedures are further examples of recursive algorithms. In each case, the recursion is guaranteed to terminate, because every recursive call involves a smaller tree, which means that we will eventually find what we are looking for or reach an empty tree.

It is clear from the algorithm that the deletion of a node requires the same number of steps as searching for a node, or inserting a new node, i.e., the average height of the binary search tree, or $O(\log_2 n)$ where n is the total number of nodes on the tree.

6.5 Checking Whether a Binary Tree Is a Binary Search Tree

What we sometimes need to do is check whether or not a given binary tree is a binary search tree, so we need an algorithm to do that. We know that an empty tree is a (trivial) binary search tree, and also that all nodes in the left subtree must be smaller than the root and themselves form a binary search tree, and all nodes in the right subtree must be greater than the root and themselves form a binary search tree. Therefore, the obvious algorithm is:

```
isbst(tree t) {
if (isEmpty(t))
   return true
else
  return (allsmaller(left(t), root(t)) and isbst(left(t))
                                    and allbigger(right(t),root(t)) and isbst(right(t)) )


allsmaller(tree t, value v) {
      if (isEmpty(t))
          return true
      else
          return ((root(t) < v) and allsmaller(left(t),v)  and allsmaller(right(t),v) )
}


allbigger(tree t, value v) {
    if (isEmpty(t))
        return true
    else
        return ((root(t) > v) and allbigger(left(t),v) and allbigger(right(t),v) )
}
```

6.6 Sorting Using Binary Search Trees

Sorting is the process of putting a collection of items in order. We shall formulate and discuss many sorting algorithms later, but we are already able to present one of them. The node values stored in a binary search tree can be printed in ascending order by recursively printing each left subtree, root, and right sub-tree in the right order as follows:

```
printInOrder(tree t) {
    if (not isEmpty(t)) {
        printInOrder(left(t))
        print(root(t))
        printInOrder(right(t))
 }
}
```

Then, if the collection of items to be sorted is given as an array a of known size n, they can be printed in sorted order by the algorithm:

```
sort(array a of size n) {
      t = EmptyTree
      for i = 0,1,...,n-1
            t = insert(a[i],t)
       printInOrder(t)
}
```

This algorithm starts with an empty tree, inserts all the items into it using insert(v,t) to give a binary search tree, and then prints them in order using printInOrder(t).

6.7 Balancing Binary Search Trees

If the items are added to a binary search tree in random order, the tree tends to be fairly well-balanced with height not much more than $\log_2 n$. However, there are many situations where the added items are not in random order, such as when adding new student IDs.

If all the items to be inserted into a binary search tree are already sorted, it is straight-forward to build a perfectly balanced binary tree from them. One simply has to recursively build a binary tree with the middle (i.e., median) item as the root, the left subtree made up of the smaller items, and the right subtree made up of the larger items. This idea can be used to rebalance any existing binary search tree, because the existing tree can easily be output into a sorted array as discussed in Section 6.6.

Another way to avoid unbalanced binary search trees is to rebalance them from time to time using tree rotations. Such tree rotations are best understood as follows: Any binary search tree containing at least two nodes can clearly be drawn in one of the two forms:

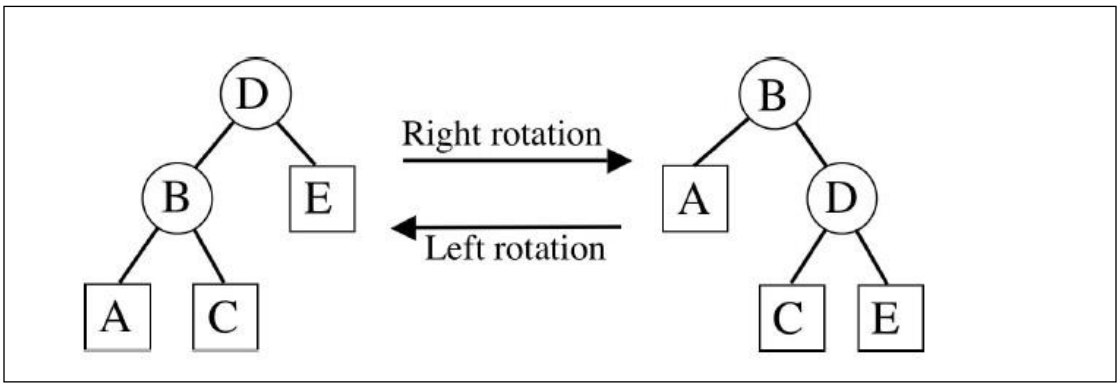


Fig. 6.2: Balancing BST Through Rotations

where B and D are the required two nodes to be rotated, and A, C and E are binary search subtrees (any of which may be empty). The two forms are related by left and right tree rotations which clearly preserve the binary search tree property. In this case, any nodes in subtree A would be shifted up the tree by a right rotation, and any nodes in subtree E would be shifted up the tree by a left rotation. For example, if the left form had A consisting of two nodes, and C and E consisting of one node, the height of the tree would be reduced by one and become perfectly balanced by a right tree rotation. In this case, the height of the tree is reduced by one because the two nodes in sub-tree A would be shifted up the tree by the right rotation.

Typically, such tree rotations would need to be applied to many different subtrees of a full tree to make it perfectly balanced. For example, if the left form had C consisting of two nodes, and A and E consisting of one node, the tree would be balanced by first performing a left rotation of the A-B-C sub-tree, followed by a right rotation of the whole tree.

6.8 B-trees

A B-tree is a generalization of a self-balancing binary search tree in which each node can hold more than one search key and have more than two children. The structure is designed to allow more efficient self-balancing, and offers particular advantages when the node data needs to be kept in external storage such as disk drives. The standard (Knuth) definition is:

A B-tree of order m is a tree which satisfies the following conditions:
- Every node has at most m children.
- Every non-leaf node (except the root node) has at least m/2 children.

- The root node, if it is not a leaf node, has at least two children.
- A non-leaf node with c children contains c - 1 search keys which act as separation values to divide its subtrees.
-All leaf nodes appear in the same level, and carry information.

The standard representation of simple order 4 example with 9 search keys would be:

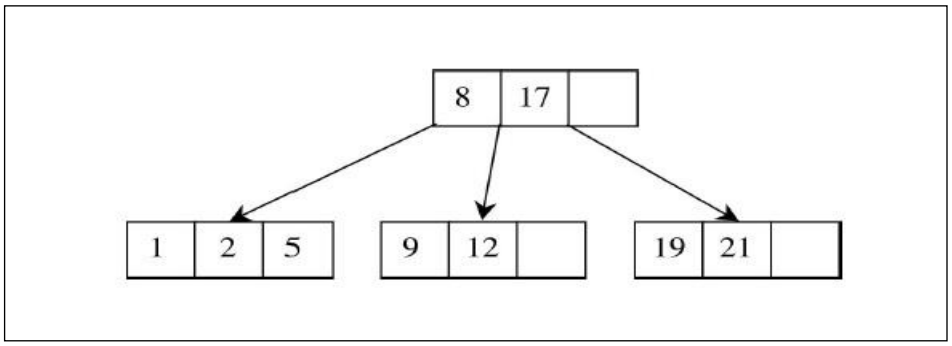


Fig. 6.3: Example of B-Tree

The search keys held in each node are ordered (e.g., 1, 2, 5 in the example), and the non-leaf node's search keys (i.e., the items 8 and 17 in the example) act as separation values to divide the contents of its subtrees in much the same way that a node's value in a binary search tree separates the values held in its two subtrees. For example, if a node has 3 child nodes (or sub-trees) then it must have 2 separation values s1 and s2.

All values in the leftmost subtree will be less than s1, all values in the middle subtree will be between s1 and s2, and all values in the rightmost subtree will be greater than s2. That allows insertion and searching to proceed from the root down in a similar way to binary search trees.

The restriction on the number of children to lie between m/2 and m means that the best-case height of an order m B-tree containing n search keys is $\log_{\mathbf{m}}n$ and the worst-case height is $\log_{\mathbf{m/2}}n$. Clearly the costs of insertion, deletion and searching will all be proportional to the tree height, as in a binary search tree, which makes them very efficient. The requirement that all the leaf nodes are at the same level means that B-trees are always balanced and thus have minimal height, though rebalancing will often be required to restore the minimal height after insertions and deletions.

C H A P T E R 7

Priority Queues and Heap Trees

7. Trees Stored in Arrays

It was noted earlier that binary trees can be stored with the help of pointer-like structures, in which each item contains references to its children. If the tree in question is a complete binary tree, there is a useful array-based alternative.

Definition. A binary tree is complete if every level, except possibly the last, is completely filled, and all the leaves on the last level are placed as far to the left as possible.

In fact, a complete binary tree is one that can be obtained by filling the nodes starting with the root, and then each next level in turn, always from the left, until one runs out of nodes. Complete binary trees always have minimal height for their size n, namely $\log_2 n$, and are always perfectly balanced (but not every perfectly balanced tree is complete in the sense of the above definition).

It is possible for them to be stored straightforwardly in arrays, top-to-bottom left-to-right, as in the following example:

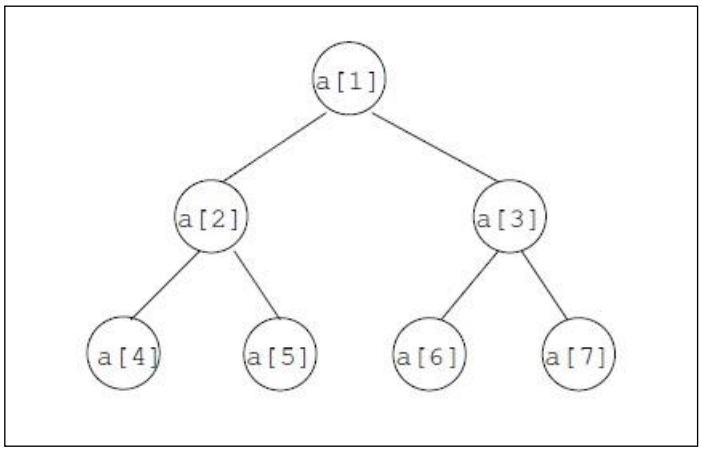


Fig. 7.1: Array-Based Binary Tree

Notice that this time we have chosen to start the array with index 1 rather than 0. This has several computational advantages.

The level of a node with index i is, $\lfloor \log_2 i \rfloor$ that is, $\log_2 i$ rounded down. The children of a node with index i, if they exist, have indices 2^i and 2^{i+1}. The parent of a child with index i has index i/2 (using integer division). This allows the following simple algorithms:

```
boolean isRoot(int i) {
return i == 1
}

int level(int i) {
return log(i)
}

int parent(int i) {
return i / 2
}

int left(int i) {
return 2^i
}

int right(int i) {
return 2^(i+1)
}
```

These make the processing of these trees much easier. This way of storing a binary tree as an array, however, will not be efficient if the tree is not complete, because it involves reserving space in the array for every possible node in the tree. Since keeping binary search trees balanced is a difficult problem, it is therefore not really a viable option to adapt the algorithms for binary search trees to work with them stored as arrays.

Array-based representations will also be inefficient for binary search trees because node insertion or deletion will usually involve shifting large portions of the array. However, we shall now see that there is another kind of binary tree for which array-based representations allow very efficient processing.

7.1 Priority Queues and Binary Heap Trees

While most queues in every-day life operate on a first come, first served basis, it is sometimes important to be able to assign a priority to the items in the queue, and always serve the item with the highest priority next. An example of this would be in a hospital casualty department, where life-threatening injuries need to be treated first. The structure of a complete binary tree in array form is particularly useful for representing such priority queues.

It turns out that these queues can be implemented efficiently by a particular type of complete binary tree known as a binary heap tree. The idea is that the node labels, which were the search keys when talking about binary search trees, are now numbers representing the priority of each item in question (with higher numbers meaning a higher priority in our examples).

With heap trees, it is possible to insert and delete elements efficiently without having to keep the whole tree sorted like a binary search tree. This is because we only ever want to remove one element at a time, namely the one with the highest priority present, and the idea is that the highest priority item will always be found at the root of the tree.

Definition: A binary heap tree is a complete binary tree which is either empty or satisfies the following conditions:
-The priority of the root is higher than (or equal to) that of its children.
-The left and right subtrees of the root are heap trees.

Alternatively, one could define a heap tree as a complete binary tree such that the priority of every node is higher than (or equal to) that of all its descendants. Or, as a complete binary tree for which the priorities become smaller along every path down through the tree.

The most obvious difference between a binary heap tree and a binary search tree is that the biggest number now occurs at the root rather than at the right-most node. Secondly, whereas with binary search trees, the left and right sub-trees connected to a given parent node play very different roles, they are interchangeable in binary heap trees.

Three examples of binary trees that are valid heap trees are:

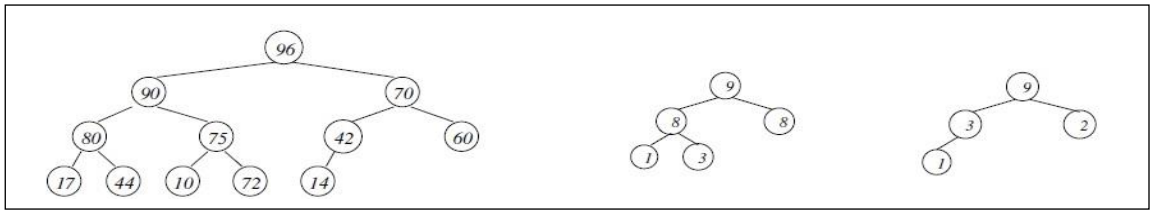


Fig. 7.2: Examples of Valid Heap Trees

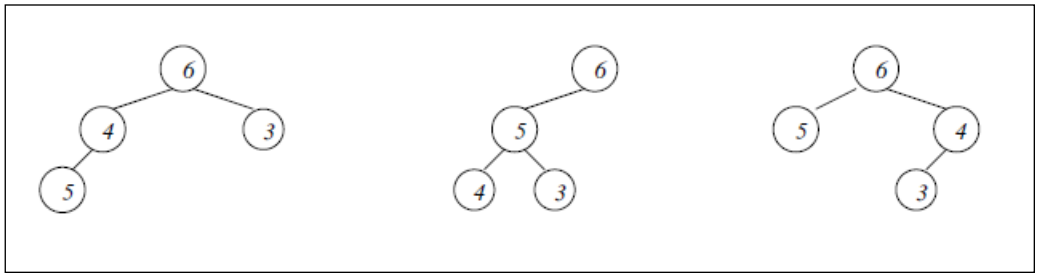


Fig.7.3: Examples of Invalid Heap Trees

The above three heap trees are invalid because in the first tree 5>4 violates the required priority ordering, the second tree because it is not perfectly balanced and hence not complete, and the third tree because it is not complete due to the node on the last level not being as far to the left as possible.

7.2 Basic Operations on Binary Heap Trees

In order to develop algorithms using an array representation, we need to allocate memory and keep track of the last position that has been filled so far, which is the same as the current number of nodes in the heap tree. The following come into play:

```
int MAX = 100         // Maximum number of nodes allowed
int heap[MAX+1]    // Stores priority values of nodes of heap tree
int n = 0             // Last position that has been filled so far
```

For heap trees to be a useful representation of priority queues, we must be able to insert new nodes (or customers) with a given priority, delete unwanted nodes, and identify and remove the top-priority node, i.e., the root (that is, `serve' the highest priority customer). We also need to be able to determine when the queue/tree is empty. Thus, assuming the priorities are given by integers, we need a constructor, mutators/selectors, and a condition:

```
insert(int p, array heap, int n)
delete(int i, array heap, int n)
int root(array heap, int n)
boolean heapEmpty(array heap, int n)
```

Identifying whether the heap tree is empty, and getting the root and last leaf, is easy:

```
boolean heapEmpty(array heap, int n) {
    return n == 0
}

int root(array heap, int n) {
    if ( heapEmpty(heap,n) )
        error(`Heap is empty')
    else return heap[1]
}

int lastLeaf(array heap, int n) {
    if ( heapEmpty(heap,n) )
        error(`Heap is empty')
```

```
    else return heap[n]
}
```

7.3 Inserting a New Heap Tree Node

Since we always keep track of the last position n in the tree which has been filled so far, we can easily insert a new element at position n + 1, provided there is still room in the array, and increment n. The tree that results will still be a complete binary tree, but the heap tree priority ordering property might have been violated. Hence, we may need to `bubble up' the new element into a valid position.

This can be done easily by comparing its priority with that of its parent, and if the new element has higher priority, then it is exchanged with its parent. We may have to repeat this process, but once we reach a parent that has higher or equal priority, we can stop because we know there can be no lower priority items further up the tree. Hence an algorithm which inserts a new heap tree node with priority p is:

```
insert(int p, array heap, int n) {
      if ( n == MAX ) {
         error(`Heap is full')
       else {
          heap[n+1] = p
          bubbleUp(n+1,heap,n+1)
        }
      }

 bubbleUp(int i, array heap, int n) {
      if ( isRoot(i) )
              return
      elseif ( heap[i] > heap[parent(i)] ) {
              swap heap[i] and heap[parent(i)]
              bubbleUp(parent(i), heap, n)
            }
       }
```

Note that this insert algorithm does not increment the heap size n; that has to be done separately by another algorithm that calls it. Inserting a node takes at most $O(\log_2 n)$ steps, because the maximum number of times we may have to `bubble up' the new element is the height of the tree which is $\log_2 n$.

7.4 Deleting a Heap Tree Node

To use a binary heap tree as a priority queue, we will regularly need to delete the root, i.e., remove the node with the highest priority. We will then be left with something which is not a binary tree at all. However, we can easily make it into a complete binary tree again by taking the node at the 'last' position and using that to fill the new vacancy at the root. However, as with insertion of a new item, the heap tree (priority ordering) property might be violated. In that case, we will need to `bubble down' the new root by comparing it with both its children and exchanging it with the largest. This process is then repeated until the new root element has found a valid place. Thus, a suitable algorithm is:

```
deleteRoot(array heap, int n) {
      if ( n < 1 )
         error(`Node does not exist')
      else {
          heap[1] = heap[n]
          bubbleDown(1,heap,n-1)
       }
}
```

A similar process can also be applied if we need to delete any other node from the heap tree, but in that case, we may need to 'bubble up' the shifted last node rather than bubble it down. Since the original heap tree is ordered, items will only ever need to be bubbled up or down, never both, so we can simply call both, because neither procedure changes anything if it is not required. Thus, an algorithm which deletes any node i from a heap tree is:

```
delete(int i, array heap, int n) {
      if ( n < i )
          error(`Node does not exist')
      else {
           heap[i] = heap[n]
           bubbleUp(i,heap,n-1)
           bubbleDown(i,heap,n-1)
       }
  }
```

The bubble down process is more difficult to implement than bubble up, because a node may have none, one or two children, and those three cases need to be handled

differently. In the case of two children, it is crucial that when both children have higher priority than the given node, it is the highest priority one that is swapped up, or their priority ordering will be violated. Thus, we have:

```
bubbleDown(int i, array heap, int n) {
        if ( left(i) > n ) // no children
                return
        elseif ( right(i) > n ) // only left child
                if ( heap[i] < heap[left(i)] )
                      swap heap[i] and heap[left(i)]
         else // two children
                if ( heap[left(i)] > heap[right(i)] and heap[i] < heap[left(i)] ) {
                      swap heap[i] and heap[left(i)]
                      bubbleDown(left(i),heap,n)
          }
                 elseif ( heap[i] < heap[right(i)] ) {
                       swap heap[i] and heap[right(i)]
                       bubbleDown(right(i),heap,n)
                  }
```

In the same way that the insert algorithm does not increment the heap size; this delete algorithm does not decrement the heap size n; that has to be done separately by another algorithm that calls it. As with insertion, deletion takes at most $O(\log_2 n)$ steps, because the maximum number of times it may have to bubble down or bubble up the replacement element is the height of the tree which is $\log_2 n$.

7.5 Building a New Heap Tree from Scratch

Sometimes one is given a whole set of n new items in one go, and there is a need to build a binary heap tree containing them. In other words, we have a set of items that we wish to heapify. One obvious possibility would be to insert the n items one by one into a heap tree, starting from an empty tree, using the $O(\log_2 n)$ `bubble up' based insert algorithm discussed earlier. That would clearly have overall time complexity of $O(n\log_2 n)$.

It turns out, however, that rearranging an array of items into heap tree form can be done more efficiently using `bubble down'. First note that, if we have the n items in an array a in positions 1, . . . ,n, then all the items with an index greater than n/2 will be leaves, and not need bubbling down. Therefore, if we just bubble down all the non-leaf

items a[n/2], . . . ,a[1] by exchanging them with the larger of their children until they either are positioned at a leaf, or until their children are both smaller, we obtain a valid heap tree.

Consider a simple example array of items from which a heap tree must be built:

5	8	3	9	1	4	7	6	2

We can start by simply drawing the array as a tree, and see that the last 5 entries (those with indices greater than 9/2 = 4) are leaves of the tree, as follows:

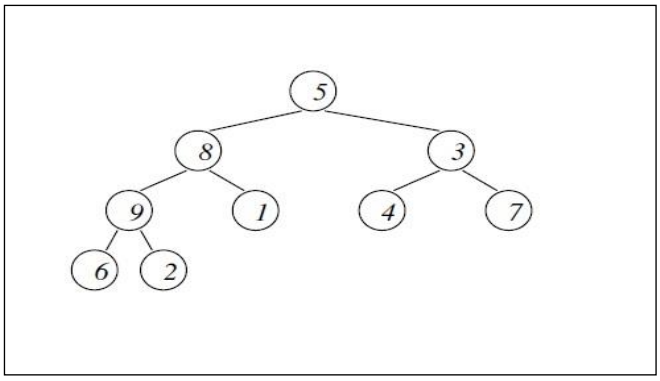


Fig.7.4: The Array Drawn as a Tree

Then the rearrangement algorithm starts by bubbling down a[n/2] = a[9/2] = a[4] = 9, which turns out not to be necessary, so the array remains the same. Next a[3] = 3 is bubbled down, swapping with a[7] = 7, giving:

5	8	7	9	1	4	3	6	2

Next, a[2] = 8 is bubbled down, swapping with a[4] = 9, giving:

5	9	7	8	1	4	3	6	2

Finally, a[1] = 5 is bubbled down, swapping with a[2] = 9, to give first index:

9	5	7	8	1	4	3	6	2

And then swapping a[2]=5 with a[4]=8 gives:

9	8	7	5	1	4	3	6	2

And finally, swapping a[4]=5 with a[8]=6 gives:

9	8	7	6	1	4	3	5	2

which has the array rearranged as the required heap tree.

Thus, using the above bubble-down procedure, the algorithm to build a complete binary heap tree from any given array a of size n is simply:

```
heapify(array a, int n) {
        for( i = n/2 ; i > 0 ; i-- )
            bubbleDown(i,a,n)
    }
```

The time complexity of this heap tree creation algorithm might be computed as follows: It potentially bubbles down ⌊n/2⌋ items, namely those with indices 1, . . . , ⌊n/2⌋ . The maximum number of bubble-down steps for each of those items is the height of the tree, which is $\log_2 n$, and each step involves two comparisons: one to find the highest priority child node, and one to compare the item with that child node. So, the total number of comparisons involved is at most $(n/2).\log_2 n.2 = n\log_2 n$, which is the same as we would have by inserting the array items one at a time into an initially empty tree. In fact, this is a good example of a situation in which a naive counting of loops and tree heights over-estimates the time complexity. This is because the number of bubble-down steps will usually be less than the full height of the tree. In fact, at each level as you go down the tree, there are more nodes, and fewer potential bubble down steps, so the total number of operations will actually be much less than $n\log_2 n$.

To be sure of the complexity class, we need to perform a more accurate calculation. At each level i of a tree of height h there will be 2^i nodes, with at most h - i bubble-down steps, each with 2 comparisons, so the total number of comparisons for a tree of height h will on average be:

$$C(h) = 2\sum_{i=0}^{h} 2^i(h-i) = 2*2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} = 2*2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} = 2*2^h \sum_{i=0}^{h} \frac{j}{2^j}$$

The final sum converges to 2^{h+1} as h increases, so for large h we have:

$$C(h) \sim 2*2^h \sum_{j=0}^{\infty} \frac{j}{2^j} = 2^h.2 = 2^{h+1} \sim n$$

and the worst case will be no more than twice that. Thus, the total number of operations is $O(2^{h+1}) = O(n)$, meaning that the complexity class of heapify is actually

O(n), which is better than the $O(n\log_2 n)$ complexity of inserting the items one at a time.

7.6 Merging Binary Heap Trees

Frequently one needs to merge two existing priority queues based on binary heap trees into a single priority queue. To achieve this, there are three obvious ways of merging two binary heap trees s and t into a single binary heap tree:

1. Move all the items from the smaller heap tree one at a time into the larger heap tree using the standard insert algorithm. This will involve moving O(n) items, and each of them will need to be bubbled up at cost $O(\log_2 n)$, giving an overall time complexity of $O(n\log_2 n)$.

2. Repeatedly move the last items from one heap tree to the other using the standard insert algorithm, until the new binary tree makeTree(0,t,s) is complete. Then move the last item of the new tree to replace the dummy root "0", and bubble down that new root. How this is best done will depend on the sizes of the two trees, so this algorithm is not totally straightforward.

On average, around half the items in the last level of one tree will need moving and bubbling, so that there will be O(n) moves, each with a cost of $O(\log_2 n)$, again giving an overall time complexity of $O(n\log_2 n)$. However, the actual number of operations required will, on average, be a lot less than the previous approach, by something like a factor of four, so this approach is more efficient, even though the algorithm is more complex.

3. Simply concatenate the array forms of the heap trees s and t and use the standard heapify algorithm to convert that array into a new binary heap tree. The heapify algorithm has time complexity O(n), and the concatenation need be no more than that, so this approach has O(n) overall time complexity, making it the best general approach of all the three.

If the two binary heap trees are such that very few moves are required for the second approach, then that may look like a better choice of approach than the third approach. However, makeTree will itself generally be an O(n) procedure if the trees are array based, rather than pointer-based, which they usually are for binary heap trees. So, for array-based similarly-sized binary heaps, the third approach is usually the best.

If the heap trees to be merged have very different sizes n and m and m < n, the first approach will have overall time complexity $O(m\log_2 n)$, which could be more efficient than an O(n) approach if m << n. In practice, a good general purpose merge algorithm would check the sizes of the two trees and use them to determine the best approach to apply.

If the value of n is large, (which it usually is, when we are considering Big O i.e., worst case), log n can be greater than 1.

So, yes, O(1) < O(log n) < O(n) < O(n log n) holds true.

C H A P T E R 8

Sorting

8. Introduction to Sorting

Arranging the data in ascending or descending order is known as sorting. The best example of sorting can be phone numbers in our phones. If, they are not maintained in an alphabetical order we would not be able to search any number effectively.

There are two types of sorting:

Internal Sorting:
If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

External Sorting:
When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, CD's or flash drives, then external sorting methods are performed.

Applications of Sorting
1. The sorting is useful in database applications for arranging the data in desired order.
2. In the dictionary-like applications, the data is arranged in sorted order.
3. For searching an element from a list of elements, sorting is required.
4. For checking the uniqueness of an element, sorting is required.
5. For finding the closest pair from a list of elements, sorting is required.

8.1 Sorting Techniques

1) Bubble sort
2) Insertion sort
3) Selection sort
4) Merge sort
5) Quick sort
5) Heap sort

8.1.1 Bubble Sort

In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

Example To discuss bubble sort in detail, let us consider an array A[] that has the following elements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 29. Since 52 > 29, swapping is done.
30, **29, 52**, 87, 63, 27, 19, 54
Compare 52 and 87. Since 52 < 87, no swapping is done.
Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87**, 27, 19, 54
Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87**, 19, 54
Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19**, **87**, 54
Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:
Compare 30 and 29. Since 30 > 29, swapping is done.
29, 30, 52, 63, 27, 19, 54, 87
Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 63. Since 52 < 63, no swapping is done.

Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, **27, 63**, 19, 54, 87
Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, **19, 63**, 54, 87
Compare 63 and 54. Since 63 > 54, swapping is done.
29, 30, 52, 27, 19, **54, 63**, 87
Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:
Compare 29 and 30. Since 29 < 30, no swapping is done.
Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, **27, 52**, 19, 54, 63, 87
Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27**, 19, 52**, 54, 63, 87
Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:
Compare 29 and 30. Since 29 < 30, no swapping is done.
Compare 30 and 27. Since 30 > 27, swapping is done.
29, **27, 30**, 19, 52, 54, 63, 87
Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, **19, 30**, 52, 54, 63, 87
Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:
Compare 29 and 27. Since 29 > 27, swapping is done.
27, 29, 19, 30, 52, 54, 63, 87
Compare 29 and 19. Since 29 > 19, swapping is done.
27, **19, 29**, 30, 52, 54, 63, 87
Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:
Compare 27 and 19. Since 27 > 19, swapping is done.
19, 27, 29, 30, 52, 54, 63, 87
Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:
Compare 19 and 27. Since 19 < 27, no swapping is done.

Observe that the entire list is sorted now.

Algorithm for Bubble Sort

BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For I = to N-1
Step 2: Repeat For J = I to N – I
Step 3: IF A[J] >A[J+ 1]
SWAP A[J] and A[J+1]
[END OF INNER LOOP]
[END OF OUTER LOOP]
Step 4: EXIT

Program

```
#include<stdio.h>
void main ()
 {
   int i, j,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   for(i = 0; i<10; i++)
   {
   for(j = i+1; j<10; j++)
  {
   if(a[j] > a[i])
   {
     temp = a[i];
     a[i] = a[j];
     a[j] = temp;
     }
```

```
   }
  }
     printf("Printing Sorted Element List ...\n");
     for(i = 0; i<10; i++)
    {
      printf("%d\n",a[i]);
     }
}
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N- 1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$f(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1$

$f(n) = n (n – 1)/2$

$f(n) = n^2/2 + O(n) = O(n^2)$

Therefore, the complexity of bubble sort algorithm is $O(n^2)$. It means the time required to execute bubble sort is proportional to n^2, where n is the total number of elements in the array.

8.1.2 Insertion Sort

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is a built-one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

Insertion sort inserts each item into its proper place in the final list. In insertion sort, the first iteration starts with comparison of 1st element with 0th element. In the second iteration 2nd element is compared with the 0th and 1st element and so on. In every iteration an element is compared with all elements. The main idea is to insert in the ith pass the ith element in A[1], A[2]…A[i] in its proper place.

Example Consider an array of integers given below. We will sort the values in the array using insertion sort:

23 15 29 11 1

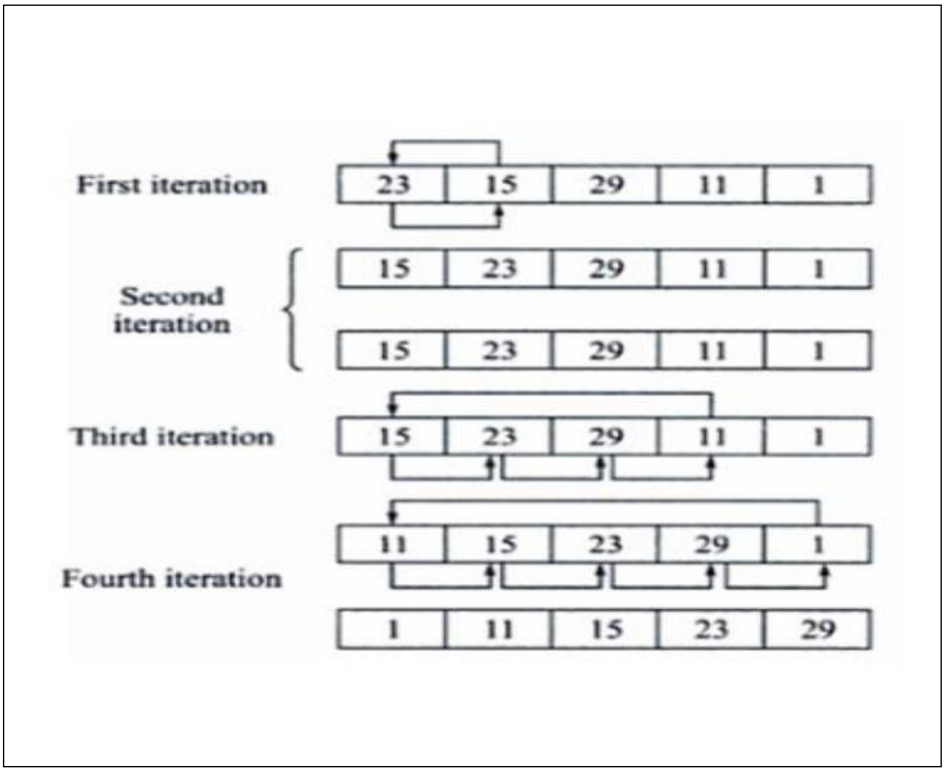


Fig. 8.1: Insertion Sort Dry Run Example

Algorithm for insertion sort

INSERTION-SORT (ARR, N)
Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

Advantages:
The advantages of this sorting algorithm are as follows:

- Relatively simple and easy to implement.
- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- The insertion sort is an in-place sorting algorithm; so, the space requirement is minimal.

Disadvantages:

- Inefficient for large list as the time complexity is $O(n^2)$.

Program

```
#include<stdio.h>
   void main ()
 {
    int i,j, k,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    printf("\nprinting sorted elements...\n");
    for(k=1; k<10; k++)
    {
       temp = a[k];
       j= k-1;
       while(j>=0 && temp <= a[j])
       {
          a[j+1] = a[j];
          j = j-1;
```

```
        }
         a[j+1] = temp;
    }
   for(i=0;i<10;i++)
   {
     printf("\n%d\n",a[i]);
    }
}
```

Output:

Printing Sorted Elements . . .

7
9
10
12
23
23
34
44
78
101

Complexity of Insertion Sort

If the initial file is sorted, only one comparison is made on each iteration, so that the sort is O(n). If the file is initially sorted in the reverse order, the worst-case complexity is $O(n^2)$.

Since the total number of comparisons is:
$(n-1)+ (n-2) +...+3+2+1= (n-1) * n/2$, which is $O(n^2)$

The average case or the average number of comparisons in the simple insertion sort is $O(n^2)$.

8.1.3 Selection Sort

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchanges it with the element in the second position, and continues in this way until the entire array is sorted.

Example: 3, 6, 1, 8, 4, 5

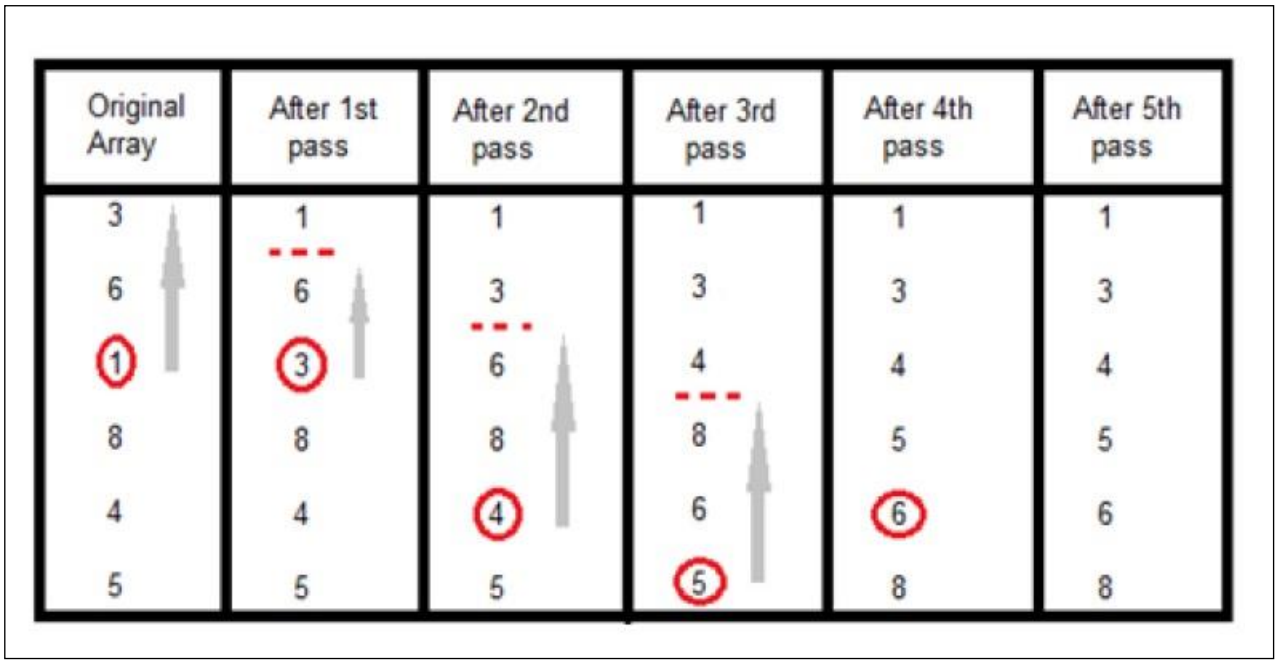


Fig. 8.2: Selection Sort Dry Run Example

Algorithm for selection sort

SELECTION SORT(ARR, N)
Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
Step 2: CALL SMALLEST(ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]
[END OF LOOP]
Step 4: EXIT

SMALLEST (ARR, K, N, POS)
Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
Step 4: RETURN POS

Advantages:

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

Disadvantages:

- Running time of Selection sort algorithm is very poor of the order of $0(n^2)$.
- However, in case of large data sets, the efficiency of the selection sort drops as
- compared to the insertion sort.

Program

```
#include<stdio.h>
 int smallest(int[],int,int);
 void main ()
 {
   int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i,j,k,pos,temp;
   for(i=0;i<10;i++)
   {
      pos = smallest(a,10,i);
      temp = a[i];
      a[i]=a[pos];
      a[pos] = temp;
    }
   printf("\nprinting sorted elements...\n");
   for(i=0;i<10;i++)
   {
     printf("%d\n",a[i]);
    }
  } //end of main()

 int smallest(int a[], int n, int i)
 {
   int small,pos,j;
   small = a[i];
   pos = i;
   for(j=i+1;j<10;j++)
   {
     if(a[j]<small)
    {
       small = a[j];
       pos=j;
```

```
      }
     }
   return pos;
  } // end of smallest()
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Complexity of Selection Sort

The first element is compared with the remaining n-1 elements in pass 1. Then n-2 elements are taken in pass 2, this process is repeated until the last element is encountered.

The mathematical expression for these iterations will be equal to:
(n-1)+(n-2)+….,+(n-(n-1))

Thus, the expression becomes n*(n-1)/2. Thus, the number of comparisons is proportional to (n^2). Therefore, the time complexity of selection sort is O(n^2).

8.1.4 Merge Sort

Merging means combining two sorted lists into one-sorted list. The merge sort splits the array to be sorted into two equal halves and each array is recursively sorted, then merged back together to form the final sorted array. The logic is to split the array into two subarrays: each subarray is individually sorted, and the resulting sequence is then combined to produce a single sorted sequence of n elements.

The merge sort recursively follows the steps:
1) Divide the given array into equal parts.

2) Recursively sorts the elements on the left side of the partitions.
3) Recursively sorts the elements on the right side of the partitions.
4) Combine the sorted left and right partitions into a single sorted array.

Example**:** Sort the array given below using merge sort.
12 ,35 ,87, 26, 9, 28, 7

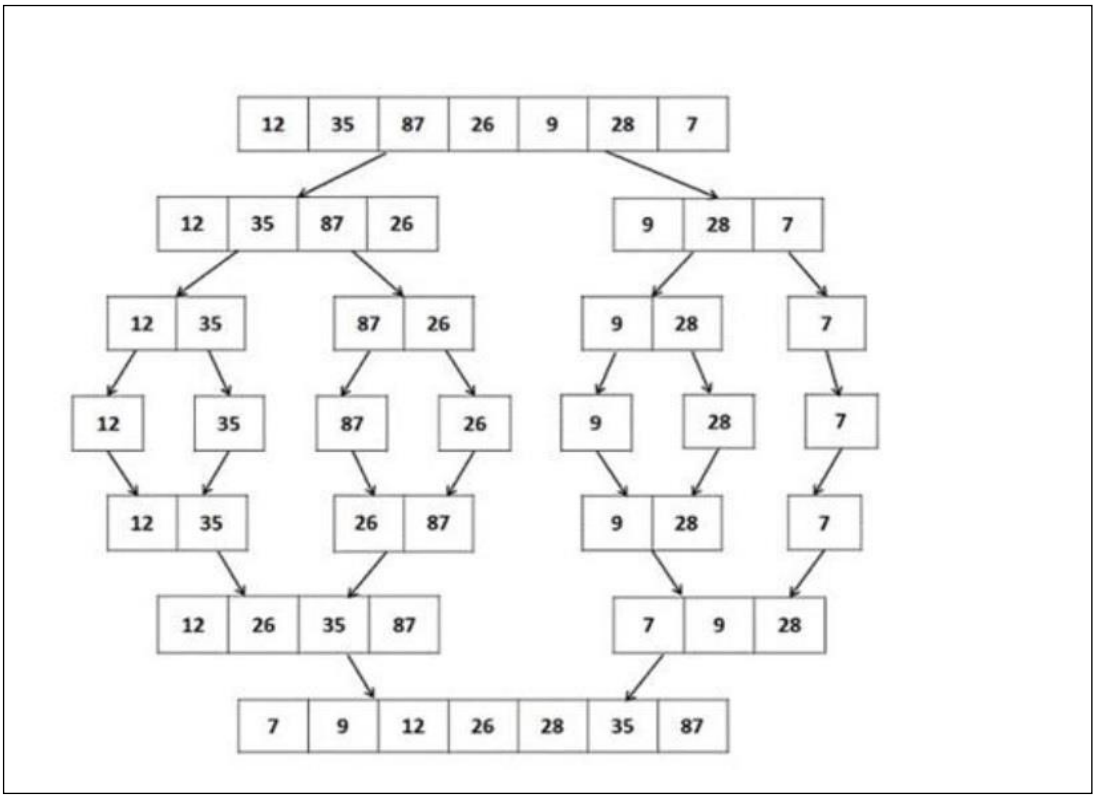


Fig. 8.3: Merge Sort Example

Algorithm for Merge Sort

MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END

SET MID = (BEG + END)/2
CALL MERGE_SORT (ARR, BEG, MID)
CALL MERGE_SORT (ARR, MID + 1, END)
MERGE (ARR, BEG, MID, END)
[END OF IF]

Step 2: END

MERGE (ARR, BEG, MID, END)

```
Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
               SET TEMP[INDEX] = ARR[I]
               SETI=I+1
         ELSE
               SET TEMP[INDEX] = ARR[J]
               SETJ=J+1
         [END OF IF]
         SET INDEX = INDEX + 1
      [END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]
         IF I>MID
           Repeat while J <= END
             SET TEMP[INDEX] = ARR[J]
             SET INDEX = INDEX + 1, SET J = J + 1
           [END OF LOOP]
           [Copy the remaining elements of left sub-array, if any]
         ELSE
            Repeat while I <= MID
                SET TEMP[INDEX] = ARR[I]
                SET INDEX = INDEX + 1, SET I = I + 1
            [END OF LOOP]
         [END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K = 0

Step 5: Repeat while K < INDEX
           SET ARR[K] = TEMP[K]
           SETK=K+1
         [END OF LOOP]

Step 6: END
```

Advantages

- Merge sort algorithm is the best case for sorting slow-access data e.g., tape drive.
- Merge sort algorithm is better at handling sequentially accessed lists.

Disadvantages

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires additional memory space of 0(n) for the temporary array .
- It goes through the whole process even if the array is sorted.

Program

```
#include<stdio.h>
 void mergeSort(int[],int,int);
 void merge(int[],int,int,int);
 void main ()
 {
    int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int i;
    mergeSort(a,0,9);
    printf("printing the sorted elements");

    for(i=0;i<10;i++)
    {
       printf("\n%d\n",a[i]);
    }
 }

 void mergeSort(int a[], int beg, int end)
{
     int mid;
     if(beg<end)
    {
       mid = (beg+end)/2;
       mergeSort(a,beg,mid);
       mergeSort(a,mid+1,end);
       merge(a,beg,mid,end);
      }
  }

 void merge(int a[], int beg, int mid, int end)
{
     int i=beg,j=mid+1,k,index = beg;
     int temp[10];
```

```
        while(i<=mid && j<=end)
        {
           if a[i]<a[j])
          {
            temp[index] = a[i];
            i = i + 1
           }

           else
           {
              temp[index] = a[j];
              j = j+1;
            }

      index++;
     }

     if(i>mid)
     {
        while(j<=end)
        {
           temp[index] = a[j];
           index++;
           j++;
         }
     }

  else
   {
      while(i<=mid)
     {
       temp[index] = a[i];
       index++;
       i++;
      }
   }
   k = beg;

   while(k<index)
  {
   a[k]=temp[k];
   k++;
   }
}
```

Output:

printing the sorted elements:

7
9
10
12
23
23
34
44
78
101

Complexity of Merge Sort

The running time of merge sort in the average case and the worst case can be given as O(nlogn). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Applications

- Merge Sort is useful for sorting linked lists in O(nlogn) time.
- Inversion Count Problem
- Used in External Sorting

8.1.5 Quick Sort

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sublists. The steps are:

1. Pick an element, called a pivot, from the list
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.
4. Apply the same method for right and left subarrays until you get a sorted array.

Example: Sort given array using Quick Sort: {2, 8, 7, 1, 3, 5, 6, 4}

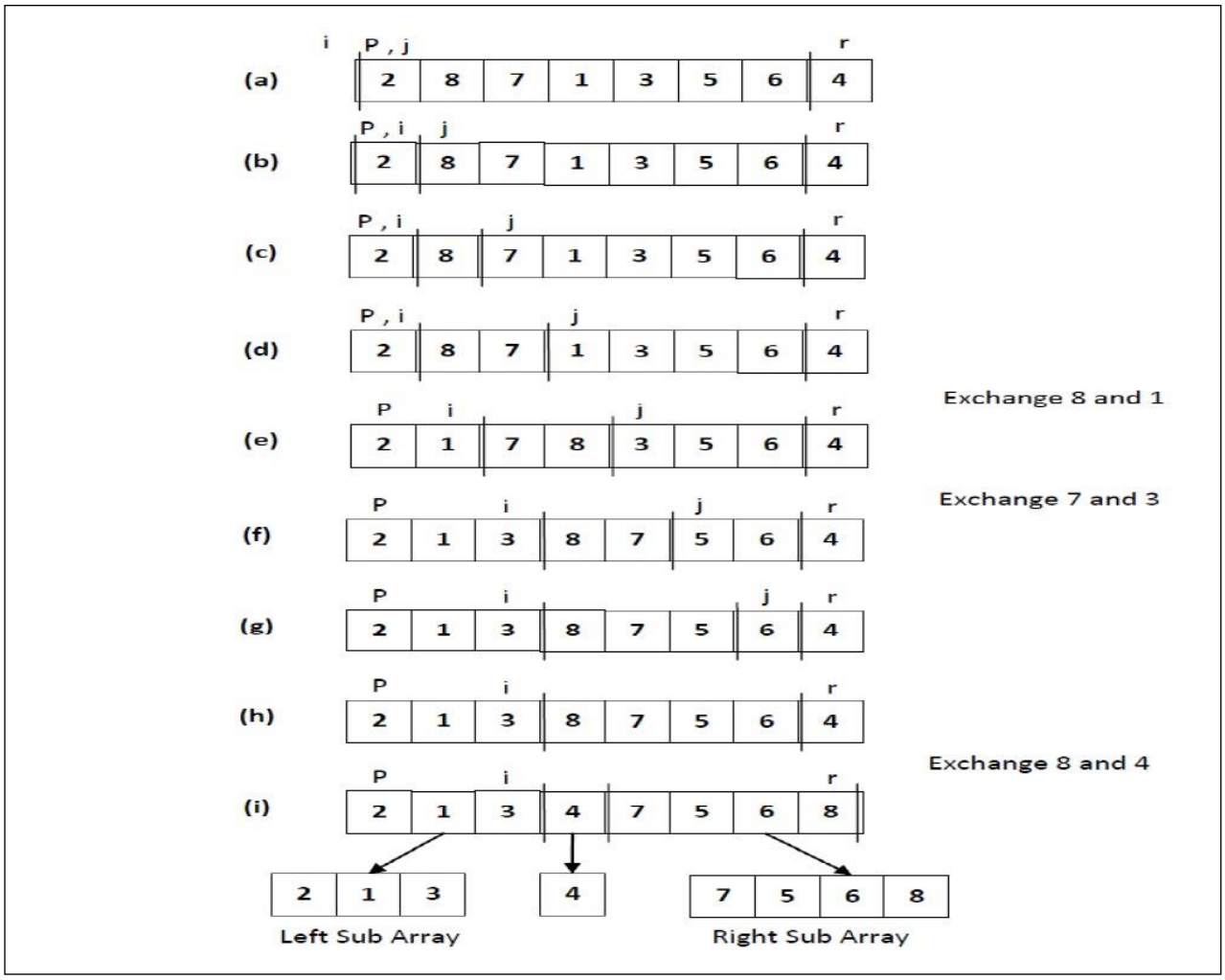


Fig. 8.4: Quick Sort Example

Algorithm for Quick Sort

QUICK_SORT (ARR, BEG, END)

```
Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0

```
Step 2: Repeat Steps 3 to 6 while FLAG = 0

Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC!= RIGHT
              SET RIGHT = RIGHT - 1
          [END OF LOOP]

Step 4: IF LOC = RIGHT
             SET FLAG = 1
          ELSE IF ARR[LOC] > ARR[RIGHT]
             SWAP ARR[LOC] with ARR[RIGHT]
             SET LOC = RIGHT
          [END OF IF]

Step 5: IF FLAG = 0
          Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
               SET LEFT = LEFT + 1
           [END OF LOOP]

Step 6: IF LOC = LEFT
              SET FLAG = 1
          ELSE IF ARR[LOC] < ARR[LEFT]
              SWAP ARR[LOC] with ARR[LEFT]
              SET LOC = LEFT
           [END OF IF]

         [END OF IF]

Step 7: [END OF LOOP]

Step 8: END
```

Advantages:

o Extremely fast $O(n\log_2 n)$

o Gives good results when an array is in random order.

o Quick sort can be used to sort arrays of small size, medium size, or large size.

Disadvantages:

o Algorithm is very complex

o In worst case of quick sort algorithm, the time efficiency is very poor which is very much similar to that of selection sort algorithm.

Program

```
1. #include<stdio.h>
2. int partition(int a[], int beg, int end);
3. void quickSort(int a[], int beg, int end);
4. void main()
5. {
6. int i;
7. int arr[10]={90,23,101,45,65,23,67,89,34,23};
8. quickSort(arr, 0, 9);
9. printf("\n The sorted array is: \n");
10. for(i=0;i<10;i++)
11. printf(" %d\t", arr[i]);
12. }

13. int partition(int a[], int beg, int end)
14. {
15.
16. int left, right, temp, loc, flag;
17. loc = left = beg;
18. right = end;
19. flag = 0;
20. while(flag != 1)
21. {
22.    while((a[loc] <= a[right]) && (loc!=right))
23.      right--;
24.    if(loc==right)
25.       flag =1;
26.    else if(a[loc]>a[right])
27.    {
28.       temp = a[loc];
29.       a[loc] = a[right];
30.       a[right] = temp;
31.        loc = right;
32.      }
33.     if(flag!=1)
34.     {
35.        while((a[loc] >= a[left]) && (loc!=left))
36.          left++;
37.        if(loc==left)
38.          flag =1;
39.        else if(a[loc] <a[left])
40.        {
41.          temp = a[loc];
42.          a[loc] = a[left];
```

```
            a[left] = temp;
            loc = left;
          }
       }
    }
return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
       loc = partition(a, beg, end);
       quickSort(a, beg, loc-1);
       quickSort(a, loc+1, end);
     }
}
```

Output:

The sorted array is:

```
23
23
23
34
45
65
67
89
90
101
```

Complexity of Quick Sort

Pass 1 will have n comparisons. Pass 2 will have 2*(n/ 2) comparisons. The subsequent passes will have 4*(n/4), 8*(n/8) comparisons and so on. The total comparisons involved in this case would be O(n)+O(n)+O(n)+⋯+s. The value of the expression will be O(n log n).Thus time complexity of quick sort is O(n log n). The space required by quick sort is very less; only O(n log n) additional space is required.

8.1.6 Heap Sort

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure. Once the heap is built, the largest and first element is at the root; so, put the

first element of the heap in an array. Then again make heap using the remaining elements to pick the first element of the heap again and put it into the array. Keep on doing the same repeatedly until we have the complete sorted list in the array.
Example: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7

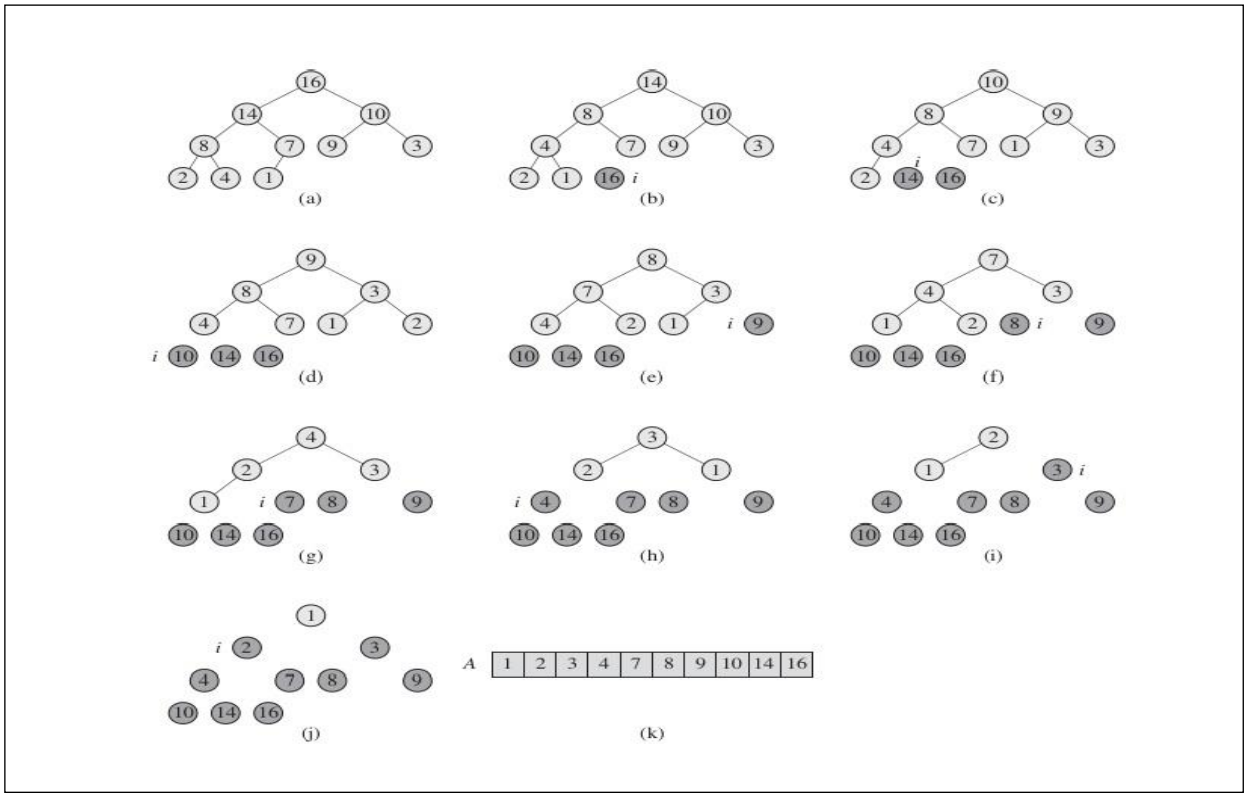


Fig. 8.5: Heap Sort Example

Operations on the Heap

Operations on the Heap, such as insertion into or deletion from the heap along with coding have been explained in detail in chapter 7. Here we review them in brief.

1. Inserting an Element into Heap
The elements are always inserted at the bottom of the original heap. After insertion, the heap remains complete but the order property is not followed so we use a bubbleUp() or heapify() operation. This involves moving the elements upward from the last position where it satisfies the order property. If the value of the last node is greater than its parent, exchange its value with that of the parent and repeat the process with each parent node until the order property is satisfied.

2. Deleting an Element from Heap
Elements are always deleted from the root of the heap.

Advantages

- Heap Sort is very fast and is widely used for sorting.
- Heap sort algorithm can be used to sort large sets of data.

Disadvantages

- Heap sort is not a stable sort, and requires a constant space for sorting a list.
- Heap sort algorithm's worst case comes with the running time of O(n log (n)) which is probably more like merge sort algorithm.

Complexity of Heap Sort

To sort an unsorted list with **'n'** number of elements, following are the complexities:

Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

C H A P T E R 9

Graphs

9. Introduction to Graphs

A Graph is a type of non-linear data structure. A map is a well-established example of a graph. In a map, various cities are connected using links. These links can be considered as roads, railway lines or aerial network.

Applications of Graphs in real life:
1. Solving Electricity Distribution problem
2. Maps like Cities, Rivers, Countries and so on
3. Water distribution in various areas
4. CAD/CAM applications
5. Finding Disaster Relief Solutions

9.1 Basic Concepts of Graphs

Nodes / Vertices: A graph contains a set of points known as nodes or vertices

Edge / Link / Arc: A link joining any two-vertex known as edge or Arc.

Graph: A graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V, E), where V is the set of vertices and E is the set of edges.

Example: Graph G can be defined as G = (V, E) where,
V = {A,B,C,D,E,F} and
E = {(A,B),(A,C),(A,D),(B,C),(C,F),(D,E),(D,F)

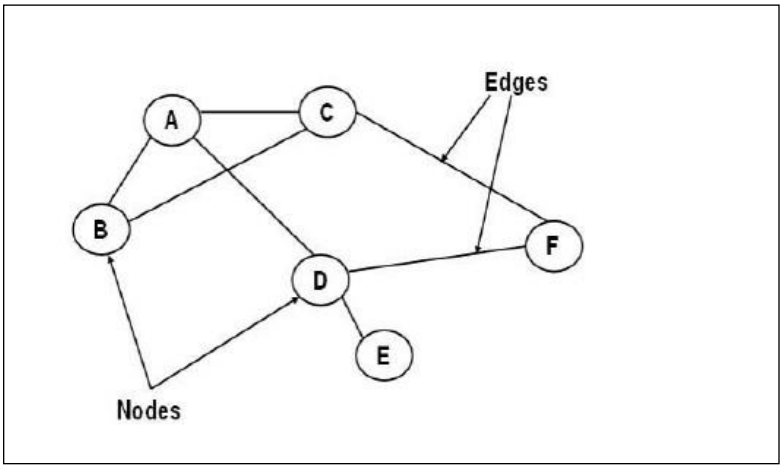


Fig. 9.1: A Graph Example

This above figure is a graph with 6 vertices and 7 edges.

Graph Terminology

1.Vertex: An individual data element of a graph is called as Vertex. Vertex is also known as node. In the above example graph, A, B, C, D, E and F are known as vertices.

2. Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).
Example: In the above graph, the link between vertices A and B is represented as (A,B).
Edges are of three types:
a. **Undirected Edge**- An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A, B) is equal to edge (B,A).

b. **Directed Edge -** A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B, then edge (A, B) is not equal to edge (B, A).

c. **Weighted Edge -** A weighted edge is an edge with cost as weight on it.

3. Degree of a Vertex: The degree of a vertex is said to the number of edges incident on it.

4.Outgoing Edge: A directed edge is said to be outgoing edge on its origin vertex.

5. Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.

6. Degree: The total number of edges connected to a vertex is said to be the degree of the vertex.

7. Indegree: The total number of incoming edges connected to a vertex is said to be indegree of that vertex.

8. Outdegree: The total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

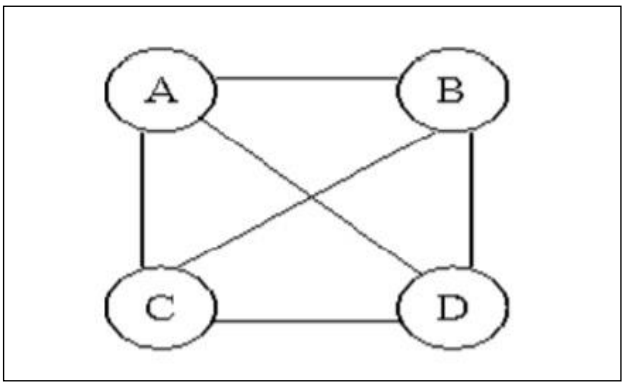
9. Parallel edges or Multiple edges: If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination, such edges are called parallel edges or multiple edges.

10. Self-loop: An edge (undirected or directed) is a self-loop if its two endpoints coincide.

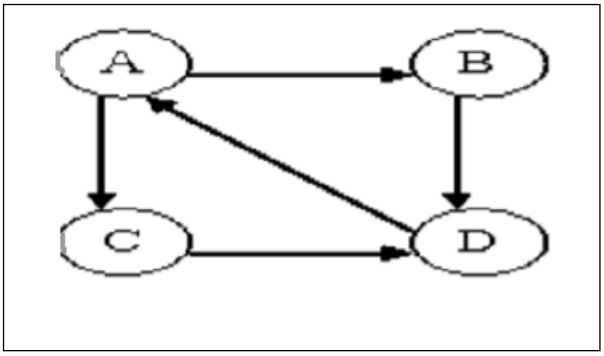
11. Simple Graph: A graph is said to be simple if there are no parallel and self-loop edges.

9.2 Types of Graphs

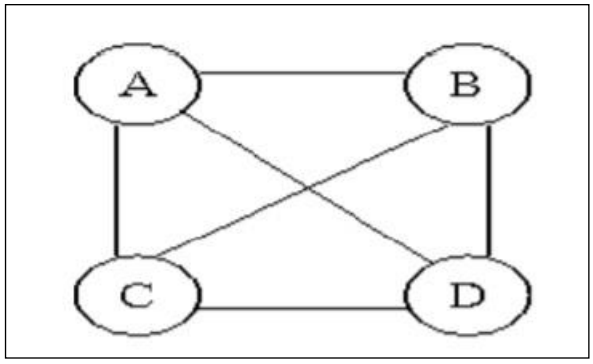
1.Undirected Graph: A graph with only undirected edges is said to be an undirected graph.



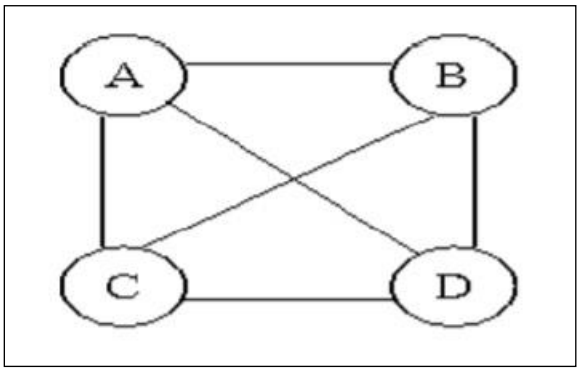
2.Directed Graph: A graph with only directed edges is said to be directed graph.



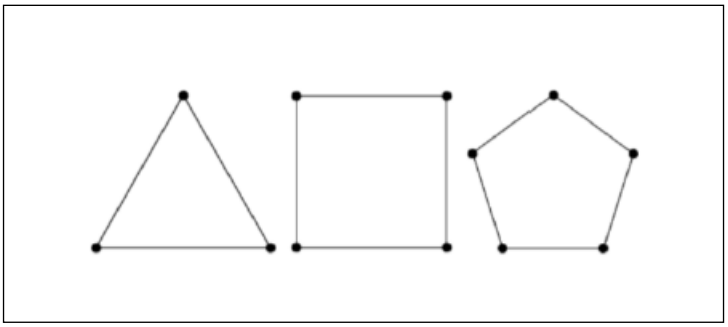
3.Complete Graph: A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the number of edges, which equals n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



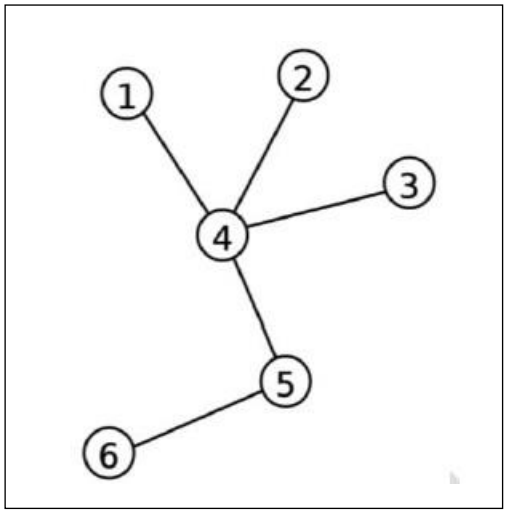
4.Regular Graph: Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



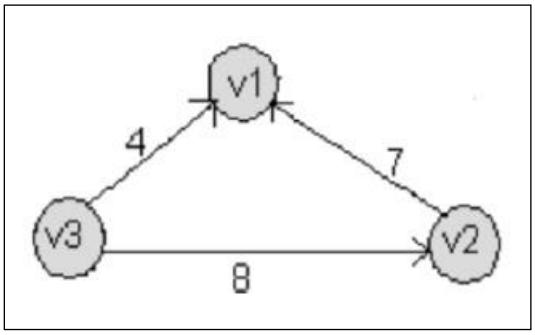
5.Cycle Graph: A graph having a cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



6.Acyclic Graph: A graph without cycle is called acyclic graphs



7. Weighted Graph: A graph is said to be weighted if there is some non-negative value assigned to each edge of the graph. The value equals the length between two vertices. A weighted graph is also called a network.



9.3 Representing Graphs

A graph data structure is represented using the following representation types:
1. Adjacency Matrix
2. Adjacency List
3. Weighted Edge

Adjacency Matrix
In this representation, a graph can be represented using a matrix of size: total number of vertices by total number of vertices; it means if a graph with 5 vertices can be represented using a matrix of 5X5 size.

In this matrix, rows and columns both represent vertices. This matrix is filled with either 1 or 0.

Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Let G = (V, E) with n vertices, n >= 1. The adjacency matrix of G is a 2-dimensional n x n matrix, A, A(i, j) = 1 iff (vi , vj) Ɛ E(G) (<vi , vj> for a graph), A(i, j) = 0 otherwise.
Example:

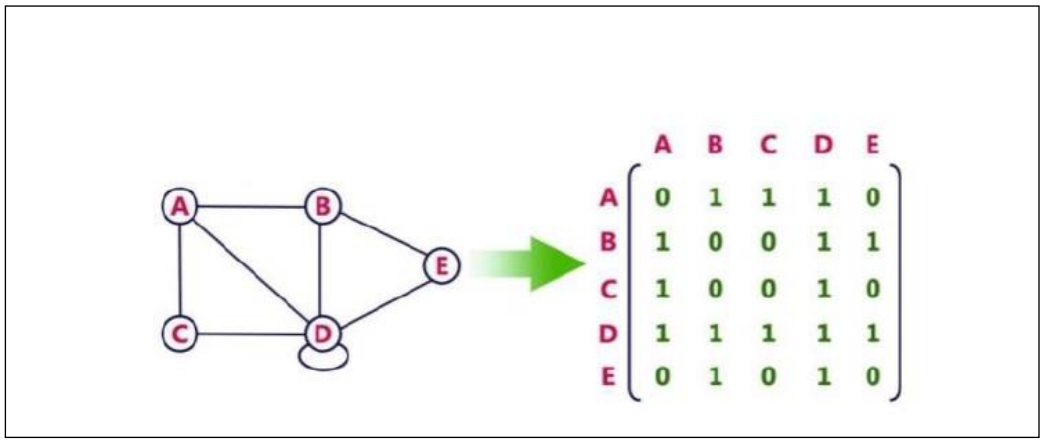


Fig. 9.2: Undirected Graph Adjacency Matrix

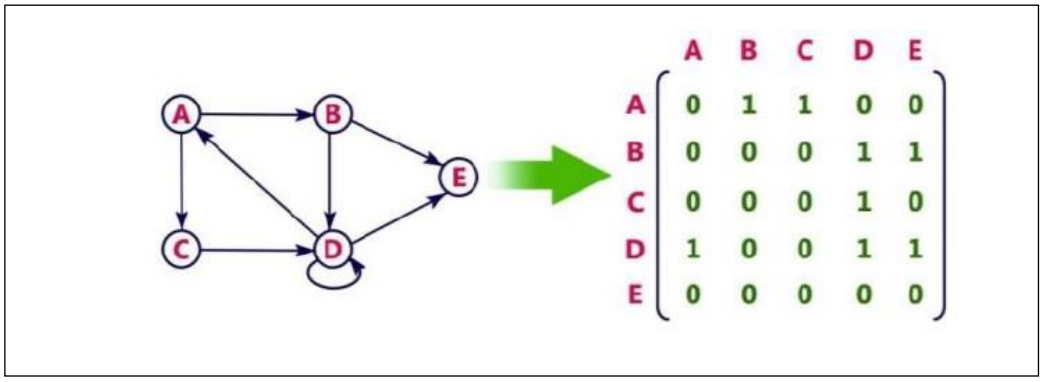


Fig. 9.3: Directed Graph Adjacency Matrix

The adjacency matrix for an undirected graph is symmetric but the adjacency matrix for a directed graph need not be symmetric.

The number of edges connected to a vertex or the degree of that vertex in an undirected graph is as follows:

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

For a directed graph, the row sum is the out_degree, while the column sum is the in_degree of a vertex.

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain i represent the vertices that are adjacent to vertex i. It can be represented in two forms.

In one form, an array is used to store n vertices and in the other form, the chain is used to store its adjacencies. Example:

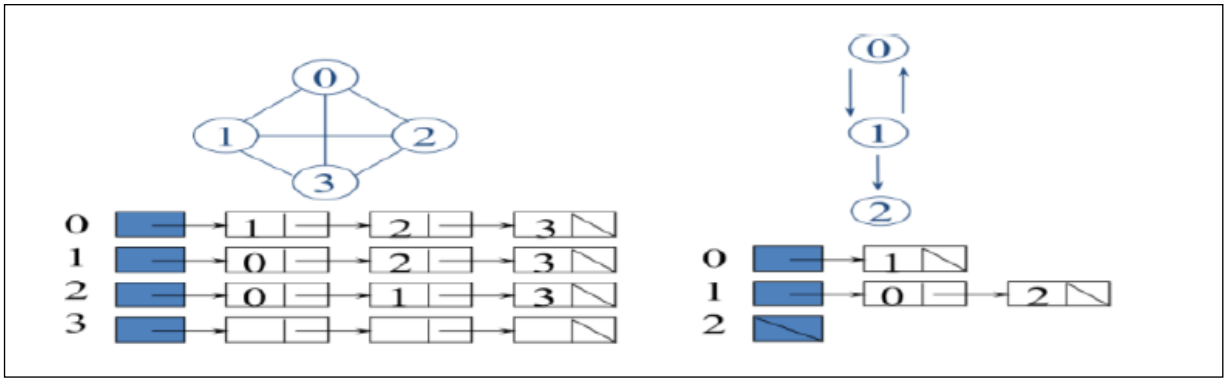


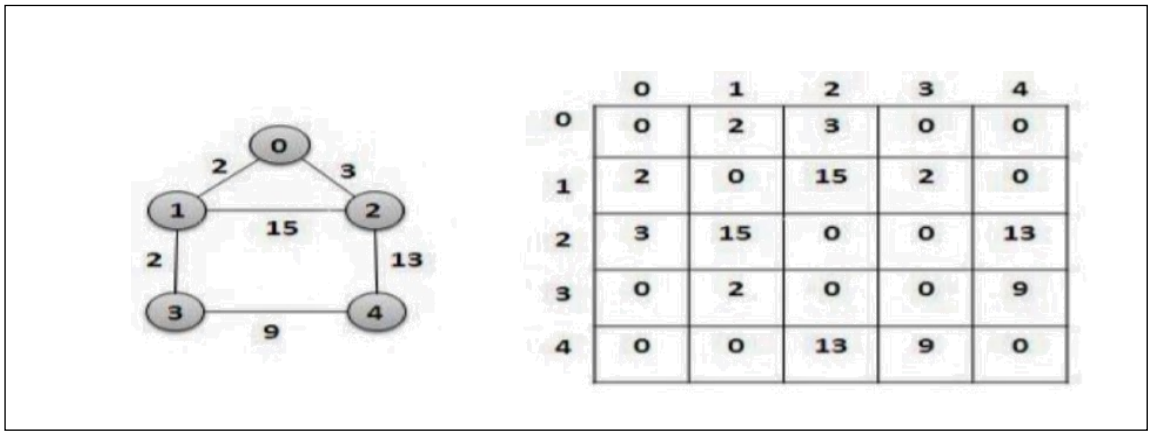
Fig. 9.4: Two forms of Representations of Adjacency Lists

We can access the adjacency list for any vertex in O(1) time.

Weighted Edge

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex.

In these applications, the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used, the weight information may be kept in the list's nodes by including an additional field weight. A graph with weighted edges is called a network.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Fig. 9.5: Adjacency Matrix Representation of Weighted Graph

9.4 Operations on Graphs

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. These methods work on both directed and undirected graphs.

Depth-First Search

The first graph operation is Depth-first search. Whenever a vertex V is visited during the search, DFS will recursively visit all of V's unvisited neighbors. Equivalently, DFS will add all edges leading out of v to a stack. The next vertex to be visited is determined by popping the stack and following that edge.

The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a depth-first search tree.

This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs. Here is an implementation for the DFS algorithm.

```
void DFS(Graph* G, int v) { // Depth first search
PreVisit(G, v); // Take appropriate action
G->setMark(v, VISITED);
for (int w=G->first(v); w<G->n(); w = G->next(v,w))
  if (G->getMark(w) == UNVISITED)
     DFS(G, w);
PostVisit(G, v); // Take appropriate action
}
```

This implementation contains calls to functions **PreVisit** and **PostVisit**. These functions specify what activity should take place during the search. Some graph traversals require that a vertex be processed before ones further along in the DFS. Alternatively, some applications require activity after the remaining vertices are processed; hence the call to function **PostVisit**.

Fig. 9.6 shows a graph and its corresponding depth-first search tree. DFS processes each edge once in a directed graph. In an undirected graph, DFS processes each edge from both directions. Each vertex must be visited, but only once, so the total cost is Θ(|V| +|E|).

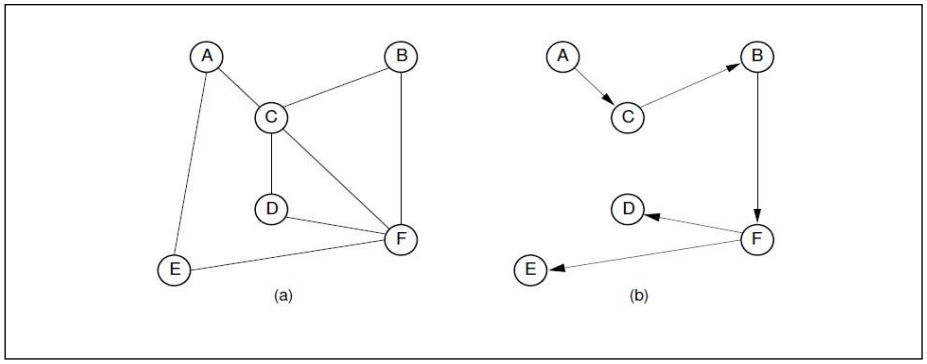


Fig. 9.6: (a) A graph (b) The depth-first search tree for the graph when starting at Vertex

Breadth First Search

Our second graph operation is known as a breadth-first search (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom. Fig. 9.7 shows a graph and the corresponding breadth-first search tree. Here is an implementation of the BFS algorithm:

```
void BFS(Graph* G, int start, Queue<int>* Q) {
int v, w;
Q->enqueue(start); // Initialize Q
G->setMark(start, VISITED);
while (Q->length() != 0) { // Process all vertices on Q
  v = Q->dequeue();
PreVisit(G, v); // Take appropriate action
for (w=G->first(v); w<G->n(); w = G->next(v,w))
  if (G->getMark(w) == UNVISITED) {
     G->setMark(w, VISITED);
Q->enqueue(w);
    }
  }
}
```

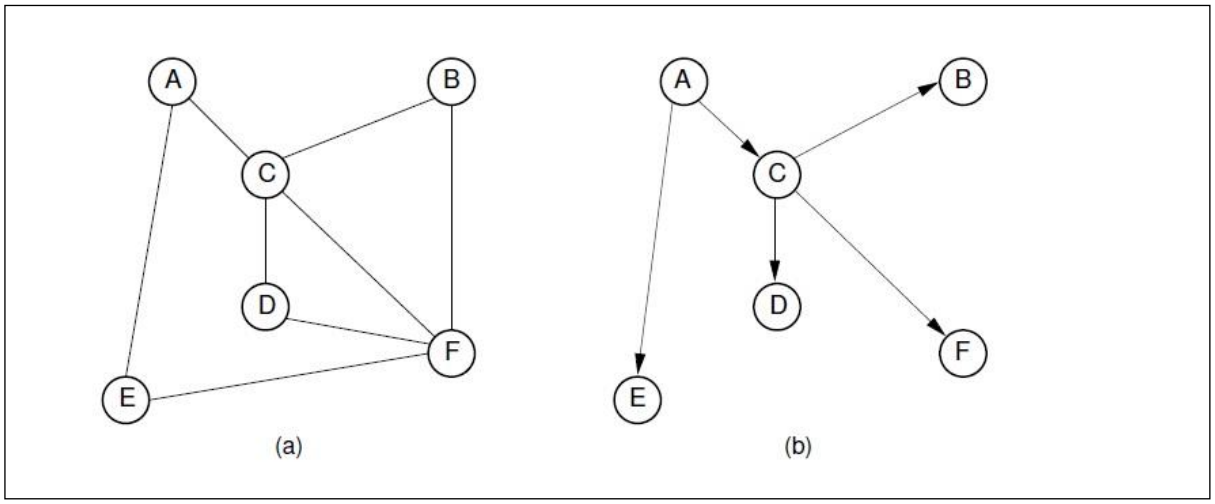


Fig. 9.7: (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex A.

9.5 Graph Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.) which have only one logical way to traverse them, trees can be traversed in different ways. The following are the generally used ways for traversing trees.

In-order

Algorithm In-order (tree)
1. Traverse the left subtree, i.e., call In-order(left subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call In-order(right-subtree)

Uses of In-order

In case of binary search trees (BST), in-order traversal gives nodes in non-decreasing order.

Example-1: In-order traversal for the below-given tree is 4 2 5 1 3

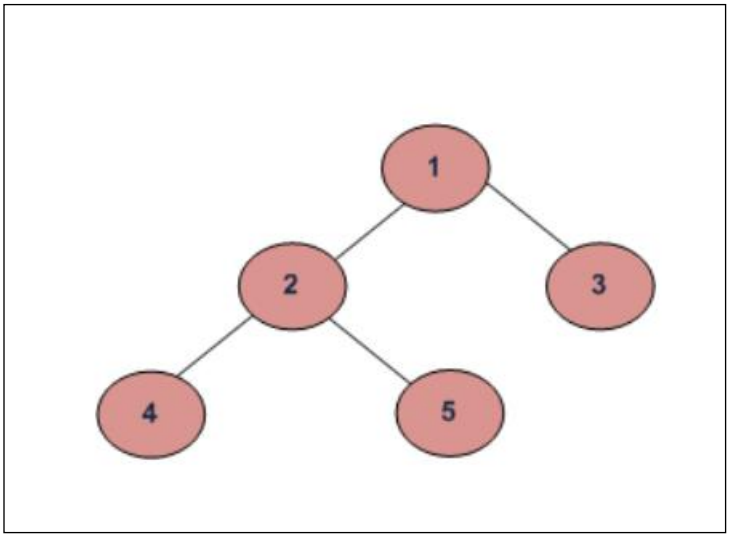


Fig. 9.8: Graph Which Can Be Traversed in In-order

Example-2: Consider Input as given below:

Input:

```
  1
 / \
3   2
```

Output: 3 1 2

Pre-order

Algorithm Pre-order (tree)
1. Visit the root.
2. Traverse the left subtree, i.e., call Pre-order(left-subtree)
3. Traverse the right subtree, i.e., call Pre-order(right-subtree)

Uses of Pre-order
Pre-order traversal is used to create a copy of the tree.

Example-1: Pre-order traversal for the below given figure is 1 2 4 5 3.

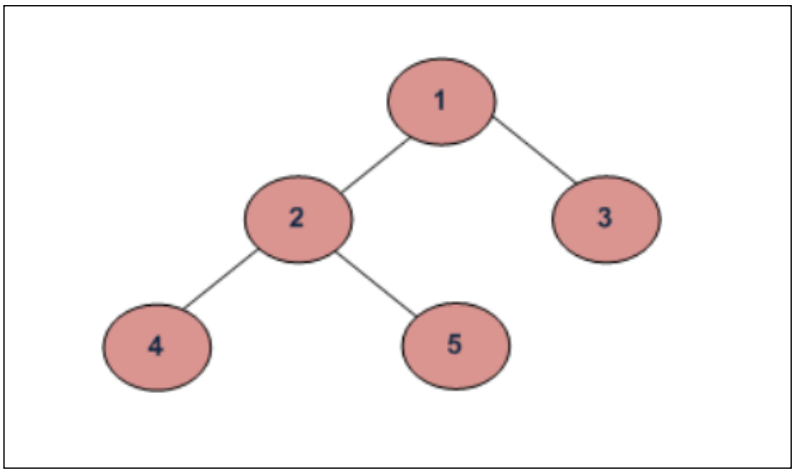
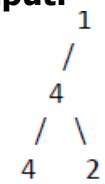


Fig. 9.9: Graph Which Can Be Traversed in Pre-order

Example-2

Input:



Output: 1 4 4 2

Post-order

Algorithm Post-order (tree)

1. Traverse the left subtree, i.e., call Post-order (left-subtree)
2. Traverse the right subtree, i.e., call Post-order (right-subtree)
3. Visit the root.

Uses of Post-order
Post-order traversal is used to delete the tree.

Example-1: Post-order traversal for the below given figure is 4 5 2 3 1

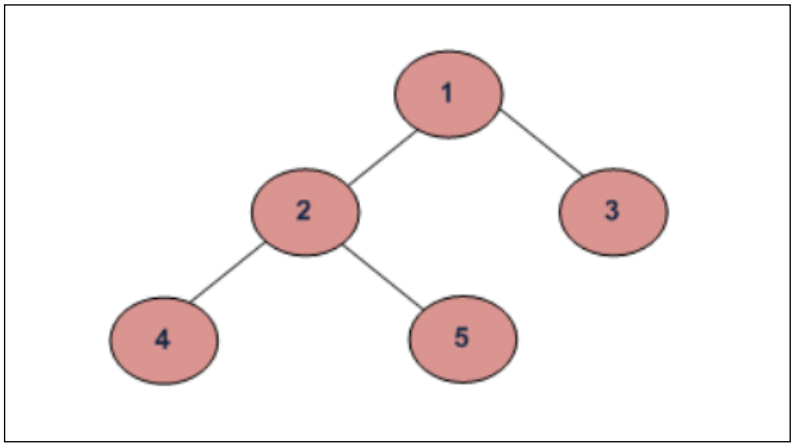
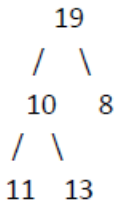


Fig. 9.10: Graph Which Can Be Traversed in Post-order

Example-2: Consider input as given below:

Input:



Output: 11 13 10 8 19

C H A P T E R 10

Graph Algorithms

10. Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

Given an undirected and connected graph G=(V, E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G).

The following figure shows the original undirected graph and its various possible spanning trees.

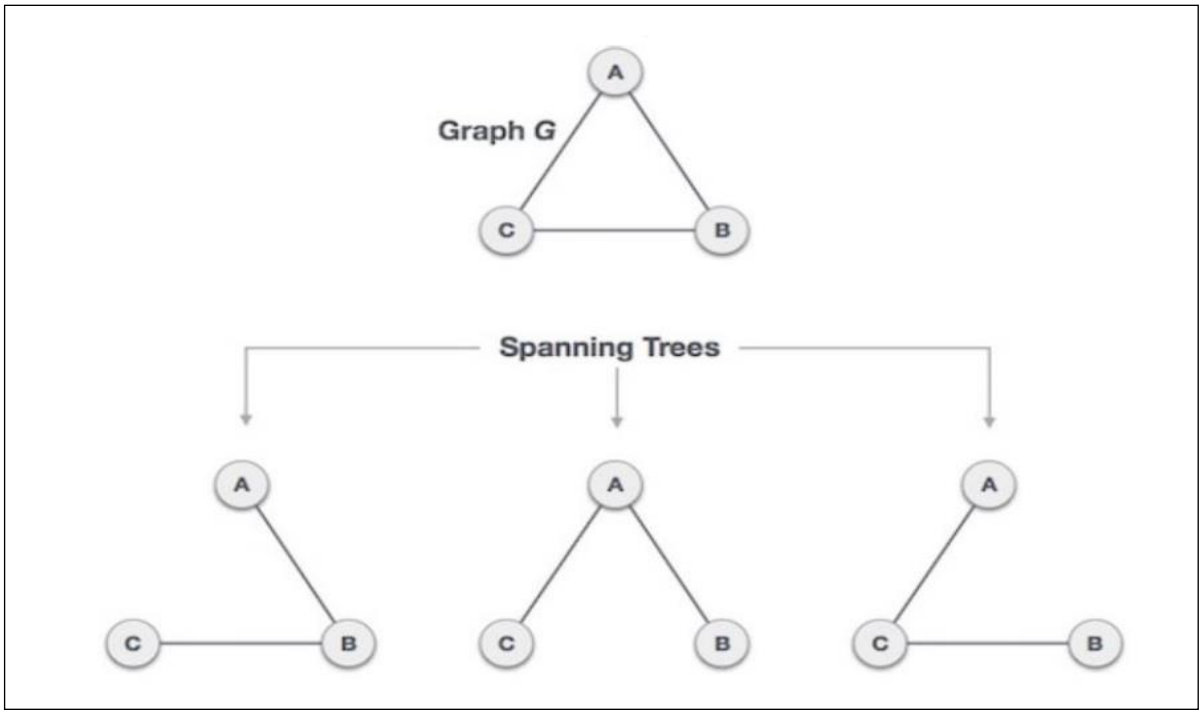


Fig. 10.1: An Undirected Graph and Its Possible Spanning Trees

We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

One graph can have more than one spanning tree. The following are a few properties of the spanning tree connected to graph G:

1. A connected graph G can have more than one spanning tree.
2. All possible spanning trees of graph G, have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).
4. Removing one edge from the spanning tree will make the graph disconnected, i.e., the spanning tree is minimally connected.
5. Adding one edge to the spanning tree will create a circuit or loop, i.e., the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

1. Spanning tree has n-1 edges, where n is the number of nodes (vertices).
2. From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
3. A complete graph can have maximum n^{n-2} number of spanning trees.
4. Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning trees.

Applications of the Spanning Tree

A spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of the spanning tree are:

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

10.1 The Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. The minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There can also be many minimum spanning trees.

The minimum spanning tree has a direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum cost weighted perfect matching. Other practical applications are:

- Cluster Analysis
- Handwriting recognition
- Image segmentation

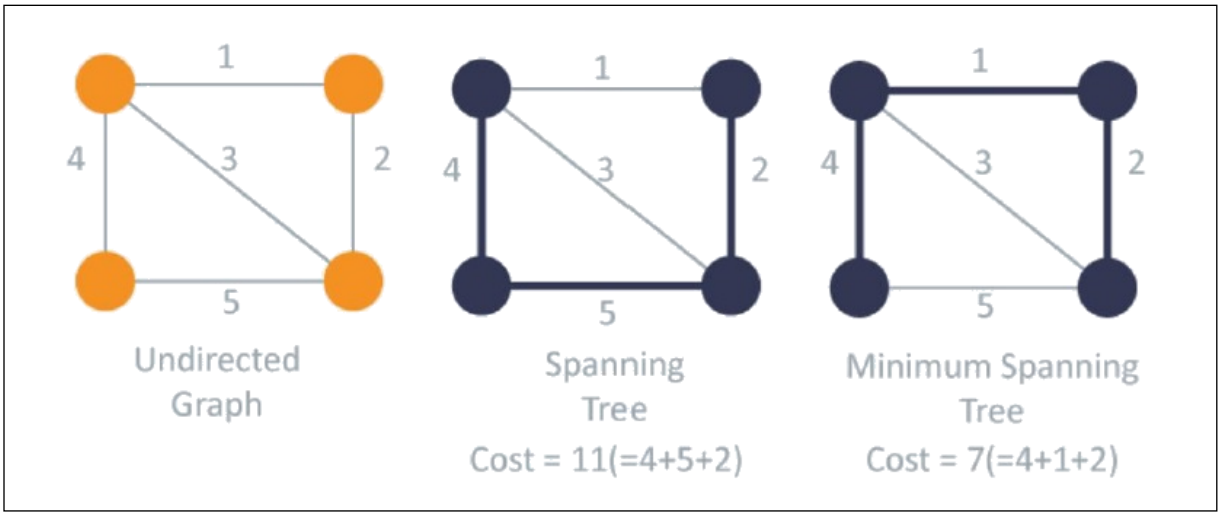


Fig. 10.2: Undirected Graph, Its Spanning Tree and Minimum Spanning Tree

There are two famous algorithms for finding the Minimum Spanning Tree:
1. Kruskal's Algorithm
2. Prim's Algorithm
Both these algorithms are Greedy Algorithms (explained in the next chapter.)

10.2 Graph Algorithms

10.2.1 Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has the least weight and adds it to the growing spanning tree.

Algorithm Steps:
1. Sort the graph edges with respect to their weights.
2. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.

3. Only add edges which don't form a cycle- edges which connect only disconnected graphs.

So now the question is how to check if 2 vertices are connected or not?
This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So, the best solution is "Disjoint Sets."

Disjoint Sets
Disjoint sets are ones whose intersection is the empty set so it means that they don't have any element in common.
Consider the following example:

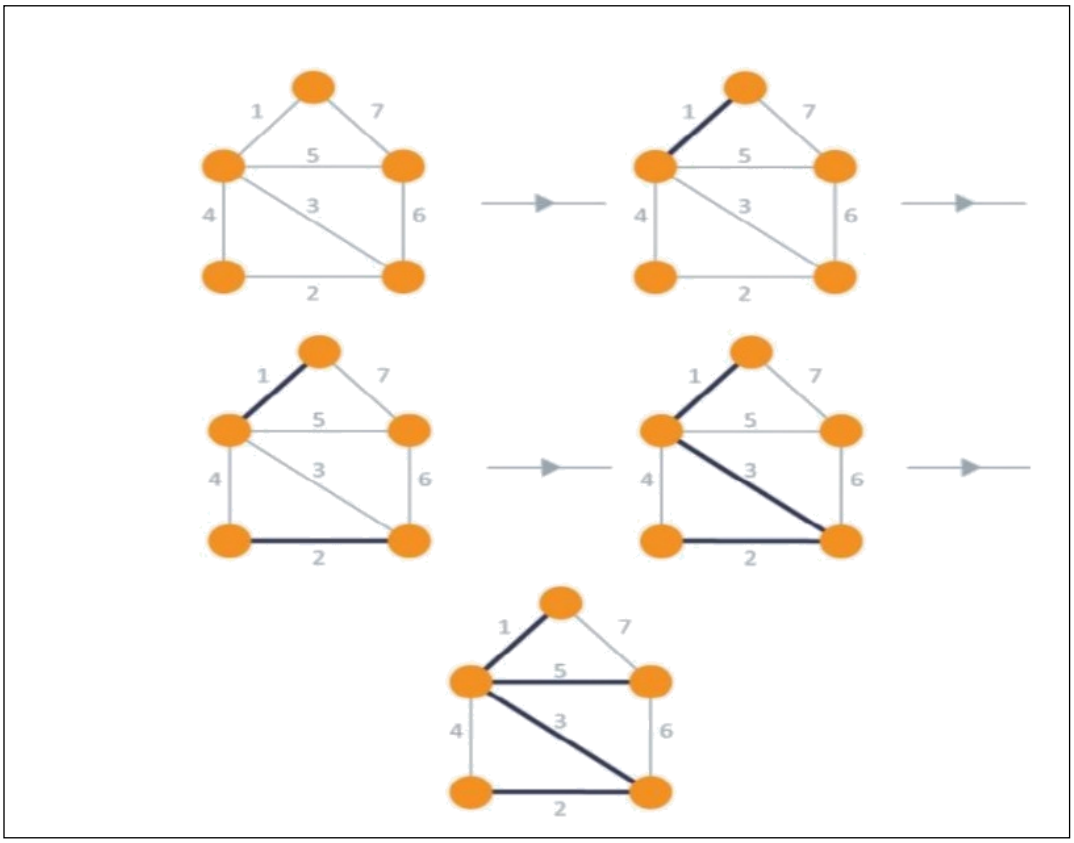


Fig. 10.3: Steps in Kruskal's Algorithm

In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, in Fig.10.4, we will start with the lowest weighted edge first i.e., the edge with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint.

Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4 that will create a cycle, and we can't have any cycles. So, we will select the fifth lowest weighted edge i.e., edge with weight 5. The other two edges will create cycles; so, we will ignore them.

In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Program:

```
#include <iostream>
#include <vector>
#include <utility>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int>> p[MAX];

void initialize()
{
for(int i = 0;i < MAX;++i)
   id[i] = i;
}

int root(int x)
{
  while(id[x] != x)
 {
  id[x] = id[id[x]];
  x = id[x];
  }
return x;
}

void union1(int x, int y)
{
int p = root(x);
int q = root(y);
id[p] = id[q];
}
```

```
long long kruskal(pair<long long, pair<int, int>> p[])
{
int x, y;
long long cost, minimumCost = 0;
for(int i = 0;i < edges;++i)
{
// Selecting edges one by one in increasing order from the beginning
x = p[i].second.first;
y = p[i].second.second;
cost = p[i].first;
// Check if the selected edge is creating a cycle or not
if(root(x) != root(y))
{
minimumCost += cost;
union1(x, y);
}
}
return minimumCost;
}

int main()
{
int x, y;
long long weight, cost, minimumCost;
initialize();
cin>> nodes >>edges;
for(int i = 0;i < edges;++i)
{
cin>> x >> y >>weight;
p[i] = make_pair(weight, make_pair(x, y));
}
// Sort the edges in the ascending order
sort(p, p + edges);
minimumCost = kruskal(p);
cout<<minimumCost<<endl;
return 0;
}
```

Time Complexity

In Kruskal's algorithm, the most time-consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(E log V), which is the overall time complexity of the algorithm.

10.2.2 Prim's Algorithm

Prim's Algorithm also uses the Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add a vertex to the growing spanning tree in Prim's.

Algorithm Steps:
1. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and others that are not in the growing spanning tree.
2. Select the cheapest vertex that is in the set of vertices of the growing spanning tree and is not in the other set of vertices. Add it to the growing spanning tree. This can be done using a Priority Queue. Insert the vertices that are connected to the growing spanning tree into the Priority Queue.
3. Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.
Consider the example below:

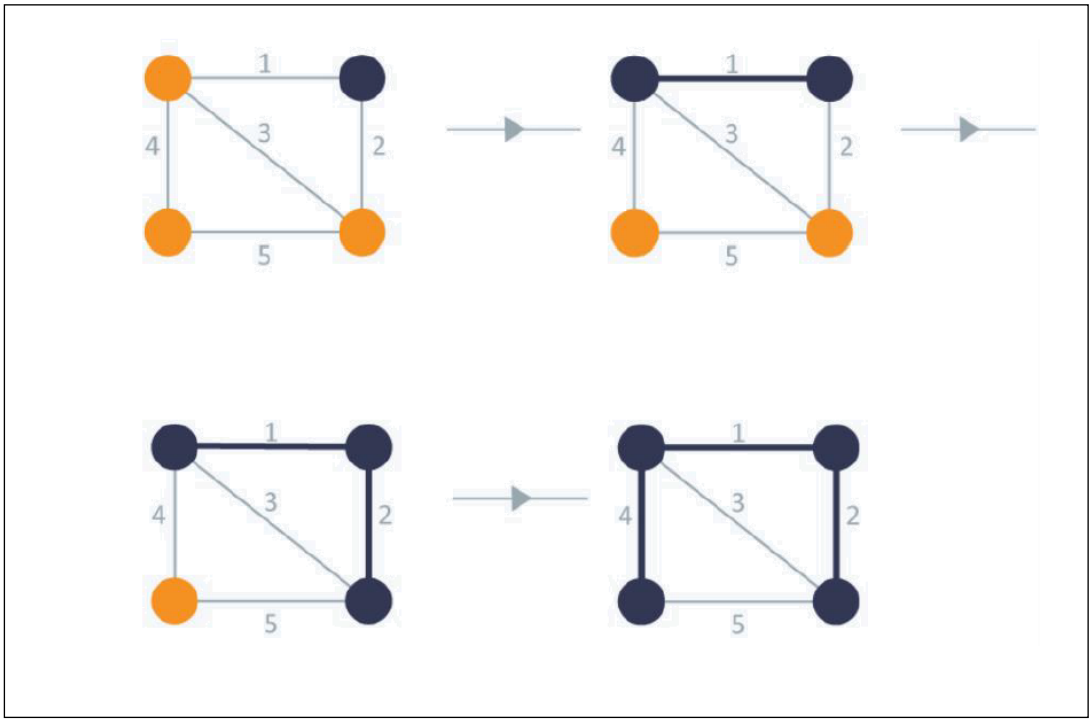


Fig. 10.4: Steps in Prim's Algorithm

In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. As we can see in Fig. 10.5, in each iteration we will mark a new vertex that is adjacent to the one that we have already marked.

As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So, we will simply choose the edge with weight 1. In the next iteration we have

three options, edges with weights 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex.

Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So, we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Program:

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>

using namespace std;
const int MAX = 1e^4 + 5;
typedef pair<long long, int>PI I;
bool marked[MAX];
vector <PI I>adj[MAX];

long long prim(int x)
{
priority_queue<PI I, vector<PI I>, greater<PI I>>Q;
int y;
long long minimumCost = 0;
PI I p;
Q.push(make_pair(0, x));
while(!Q.empty())
{
// Select the edge with minimum weight
p = Q.top();
Q.pop();
x = p.second;
// Checking for cycle
if(marked[x] == true)
continue;
minimumCost += p.first;
marked[x] = true;
for(int i = 0;i <adj[x].size();++i)
{
y = adj[x][i].second;
if(marked[y] == false)
Q.push(adj[x][i]);
}
```

```
}turn minimumCost;
}

int main()
{
int nodes, edges, x, y;
long long weight, minimumCost;
cin>> nodes >>edges;
for(int i = 0;i < edges;++i)
{
cin>> x >> y >>weight;
adj[x].push_back(make_pair(weight, y));
adj[y].push_back(make_pair(weight, x));
}
// Selecting 1 as the starting node
minimumCost = prim(1);
cout<<minimumCost<<endl;
return 0;
}
```

Time Complexity

The time complexity of the Prim's Algorithm is O((V+E) log V) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

C H A P T E R 11

Algorithm Design Techniques

11. Introduction

Programmers and developers build apps, puzzle games, and video games and also, program shortest distance travels among places and regions, resulting in minimum cost. For these to achieve, they need to base their programs on optimal and valid algorithms. Here is where algorithm design techniques come into the picture.

I have considered here four main algorithm design strategies that are easy to follow. You need to follow an algorithm design technique that needs to match a problem appropriately, making use of the algorithm with good time complexity.

Not all algorithms have the same time complexity, or even different algorithms for a particular application may have the same time complexity but still differ in their performance.

Among the four main algorithmic design techniques that I cover, I explain them along with examples of applications for which they are suitable as well as their advantages and disadvantages.

I make comparisons among the techniques based on what common specific features they have, which problems they commonly solve well, and the time complexities of the corresponding algorithms for the problems. I make the survey in easy, tabular formats.

11.1 What Are Algorithm Design Techniques?

An Algorithm Design Technique is an approach for creating algorithms and solving problems. Multiple algorithms can solve a problem but not all of them can do so efficiently. Here are five algorithm design techniques (we will not be using Brute Force method) that we will discuss in this chapter.

- Brute Force Method
- Divide and Conquer Strategy

- Backtracking Method
- Dynamic Programming
- Greedy Method

11.2. Objectives

A survey of five different algorithm design techniques is done to infer which of the algorithms are efficient for which type of applications. Sometimes some of the techniques overlap for some of the problems, and again, a comparison is made among the algorithms to find out which one of them is really performing well to give the best solution and whether there are trade-offs among them.

11.3 Brute Force Method (BF)

A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found. The time complexity of a brute force algorithm is often proportional to the input size. Brute force algorithms are simple and consistent, but very slow.

Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

Application of the Brute Force Method

If there is a lock of 4-digit PIN and the digits to be chosen are from 0-9, then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 tries to find the right combination.

We will not linger on the brute force approach as it is a slow method.

11.4 Divide and Conquer Strategy (D&Q)

In divide and conquer approach, the original problem is broken down into subproblems. These are like the original problem but smaller in size and simpler to solve. The

subproblems are solved recursively, and their solutions are combined to give the solution to the original problem.

At each level of the recursion, the divide and conquer approach follows three steps:
1) Divide: In this step, the whole problem is divided into several subproblems.

2) Conquer: The subproblems are conquered by solving them recursively, only if they are small enough to be solved, otherwise step1 is executed.

3) Combine: In this final step, the solutions obtained by the subproblems are combined to create the solution to the original problem.

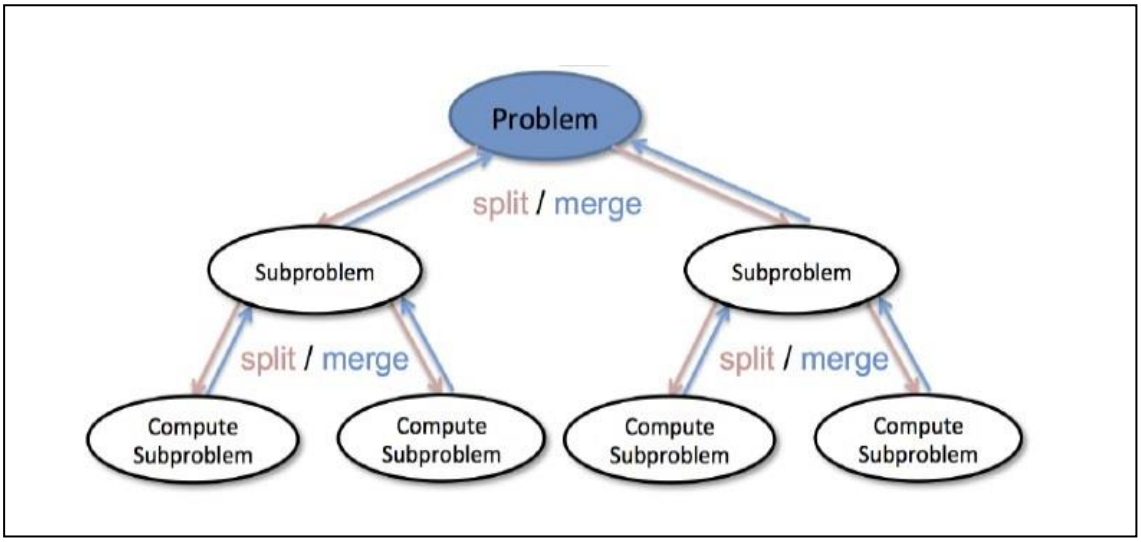


Fig. 11.1: Divide and Conquer Approach

Applications of Divide and Conquer Approach

The following algorithms are based on the concept of the Divide and Conquer technique:

1) Binary Search
2) Quicksort
3) Merge Sort
4) Maximum and Minimum Problem
5) Towers of Hanoi

Advantages of Divide and Conquer

1) Divide and conquer tends to successfully solve one of the biggest problems, such as the Towers of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively.

If you take something that grows quadratically and cut it into two pieces, each of which is half the size as before, it takes one quarter of the initial time to solve the problem in each half, so solving the problem in both halves takes time roughly one half the time required for the brute force solution.

2)It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.

Disadvantages of Divide and Conquer

1) Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.

2) An explicit stack may overuse the space.

3) It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

11.5 Backtracking (BT)

The Backtracking is another algorithmic method to solve a problem. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combinations to solve an optimization problem.

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

In the following figure:

1) Each non-leaf node in a tree is a parent of one or more other nodes (its children)
2) Each node in the tree, other than the root, has exactly one parent.

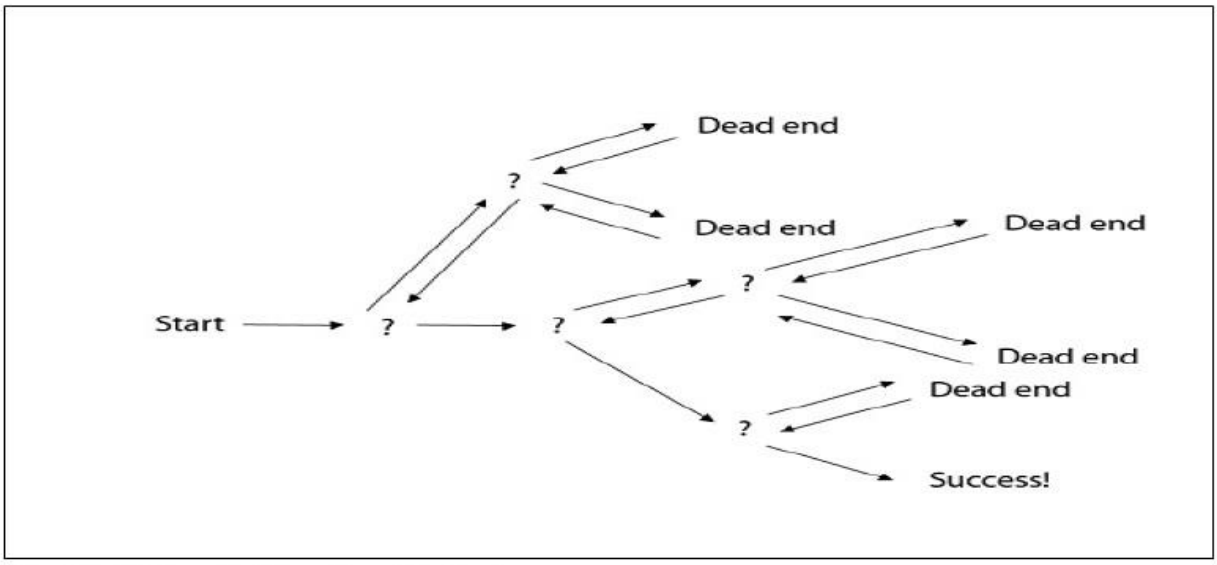


Fig. 11.2: The Backtracking Method

A 'Goal' can be understood of as searching a tree for a particular "goal" leaf node. Nodes represent solutions, and non-goal terminal nodes represent dead ends.

The goal node is said to be a success node if it provides a feasible solution. A dead node is the node which cannot be further generated and also does not provide a feasible solution.

Backtracking is undoubtedly quite simple - we "explore" each node, as follows:

To "explore" node N:
1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,
Explore C
If C was successful, return "success"
4. Return "failure"

Code 11.1: Demonstrating Backtracking

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a depth-first search with any bounding function. All solutions using backtracking are needed to satisfy a complex set of constraints.

Applications of the Backtracking Algorithm

Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for parsing, for the knapsack problem and other combinatorial optimization problems.

Advantages of Backtracking

1) It considers searching every possible combination in order to solve a computational problem. In Decision Problem, we search for a feasible solution. In Optimization Problem, we search for the best solution.

2) It avoids the need to generate and evaluate every potential solution.

Disadvantages of Backtracking

1) It can disqualify a lot of conflicting variables with just one test.

2) The other drawback of backtracking is having to perform redundant work. Even if the conflicting values of variables are identified during the intelligent backtracking, they are not remembered for immediate detection of the same conflict in subsequent computations.

11.6 Dynamic programming (DP)

Dynamic Programming Technique is like the divide-and-conquer technique. Both the techniques solve a problem by breaking it down into several sub-problems that can be solved recursively. The main difference between them is that Divide & Conquer approach partitions the problems into independent sub-problems, solve the sub-problems recursively, and then combine their solutions to solve the original problem whereas dynamic programming is applicable when the sub-problems are not independent, that is, when sub-problems share sub subproblems.

Also, a dynamic programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub-subproblem is encountered. Therefore, **"**Dynamic programming is applicable when subproblems are not independent, that is when subproblems share subproblems.**"**

As with the Greedy approach described next, Dynamic programming is typically applied to optimization problems and for them, there can be many possible solutions and the requirement is to find the optimal solution among those. But Dynamic programming approach is a little different from the greedy approach.

Greedy solutions are computed by making choices in the serial forward way, and in this, no backtracking & revision of choices is done whereas Dynamic programming computes its solution bottom up by producing them from smaller subproblems, and by trying many possibilities and choices before it arrives at the optimal set of choices.

The Development of a dynamic programming algorithm can be broken into a sequence of four steps:

Divide, Subproblems: The main problems are divided into several smaller overlapping subproblems. In fact, the solution of the main problem is expressed in terms of the solutions for the smaller subproblems. Basically, it is all about characterizing the structure of an optimal solution and recursively defining the value of an optimal solution.

Table, Storage**:** The solution for each subproblem is stored in a table, so that it can be used many times whenever required.

Combine, Bottom-up Computation: The solution to the main problem is obtained by combining the solutions of smaller subproblems to solve the larger subproblems. i.e., computing the value of an optimal solution in a bottom-up fashion.

Optimal Solution: Construct an optimal solution from computed information.

Now for any problem to be solved through dynamic programming approach it must follow the following conditions:

Principle of Optimality: It states that for solving the master problem optimally, its subproblems should be solved optimally. It should be noted that not all the time, each subproblem is solved optimally; so, in that case we should go for optimal majority.

Polynomial Breakup: For solving the main problem, the problem is divided into several subproblems, and for efficient performance of dynamic programming, the total number of subproblems to be solved should be at most a polynomial number.

Applications of Dynamic Programming

Various algorithms which make use of the Dynamic programming technique are as follows:

1. Knapsack problem.
2. Chain matrix multiplication.
3. All pair shortest path.
4. Travelling salesman problem.
5. Towers of Hanoi.
6. Checker Board.
7. Fibonacci Sequence.
8. Assembly line scheduling.
9. Optimal binary search trees.

Advantages of Dynamic Programming

1) Dynamic programming is useful for solving problems that involves breaking down problems into smaller overlapping sub-problems, storing the results computed from the sub-problems and reusing those results on larger chunks of the problem.

2) The main use of dynamic programming is to solve optimization problems.

3) It is very easy to understand and implement.

4) It solves the subproblems only when it is required.

5) It is easy to debug.

Disadvantages of Dynamic Programming

1)Dynamic programming uses recursion, which requires more memory in the call stack, and leads to a stack overflow condition in the runtime.

2)It takes up memory to store the solutions of each subproblem. There is no guarantee that the stored value will be used later in execution.

11.7 Greedy Methods (GM)

It is hard, if not impossible, to define precisely what is meant by a greedy algorithm. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to maximize some underlying criterion. One can often design many

different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

When a greedy algorithm succeeds in solving a non-trivial problem optimally, it typically implies something interesting and useful about the structure of the problem itself; there is a local decision rule that one can use to construct optimal solutions. The same is true of problems in which a greedy algorithm can produce a solution that is guaranteed to be close to optimal, even if it does not achieve the precise optimum. It's easy to invent greedy algorithms for almost any problem, finding cases in which they work well, and proving that they work well, is the interesting challenge.

We can develop two basic methods for proving that a greedy algorithm produces an optimal solution to a problem. One can view the first approach as establishing that the greedy algorithm stays ahead. By this we mean that if one measures the greedy algorithm's progress in a step-by-step inductive fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution.

The second approach is known as an exchange argument, and it is more general; one considers any possible solution to the problem, and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will follow that the greedy algorithm must have found a solution that is at least as good as any other solution.

Applications of the Greedy Method

Following this introduction of the above two styles of analysis, there are two of the most well-known applications of greedy algorithms: shortest paths in a graph, and the minimum spanning tree problem.

Advantages of the Greedy Method

1) Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and Conquer).

2) The greedy approach is easy to implement.

3) Typically, they have less time complexity.

4) Greedy algorithms can be used for optimization purposes or finding close to optimization in case of NP-Hard problems.

Note: A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP-problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem," although it might, in fact, be harder.

For example, the optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph, commonly known as the travelling salesman problem, is NP-hard.

Disadvantages of the Greedy Method

1) The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues.

2) The biggest drawback involved with making use of greedy algorithms is that it is possible that the local optimal solution might not always be the global optimal solution.

3) The greedy method might not provide the optimal solution every time.

11.8 Comparisons Among the Algorithmic Techniques

Looking closely at the concepts, features, applications, advantages, and disadvantages of the four main algorithm design techniques, we make valuable comparisons in tabular formats and state the findings.

Table 11.1. Comparisons of Features Among Main 4 Algorithm Design Techniques

	Features			
	Split/Merge	**Optimal Solutions**	**No Backtracking/ Review Choices**	**Must Have Polynomial Number of Subproblems**
D&Q	✓		✓	
BT		✓		
DP	✓	✓		✓
GM		✓	✓	

As we can see from Table 11.1, both Divide and Conquer strategy and Dynamic Programming method use the concept of split and merge feature to arrive at a solution.

Backtracking, Dynamic Programming and Greedy Methods use the feature of optimal solutions for solving algorithmic problems. We will soon come to the part about which algorithmic problems they solve efficiently.

Divide and Conquer strategy and Greedy Method use no backtracking or reviewing choices concept.

Dynamic Programming technique is the only one among the four main ones to have the requirement of polynomial number of subproblems.

Table 11.2. Comparisons of Efficient Applications Among 4 Main Algorithm Design Techniques

	Common Applications		
	Binary Search/ Towers of Hanoi	**Binary Knapsack Problem**	**Shortest Paths in a Graph**
D&Q	✓		
BT		✓	
DP	✓	✓	✓
GM			✓

In Table 11.2, we find that Divide and Conquer strategy and Dynamic Programming technique are commonly good for solving applications such as binary search and Towers of Hanoi.

Again, we observe that Backtracking and Dynamic Programming techniques are both good for solving the binary knapsack problem. Between the two, which one is even better, we shall soon discuss.

The problem of shortest paths in a graph can be well handled by Dynamic Programming technique and Greedy Method.

It can be seen that Dynamic Programming along with another technique among the remaining three commonly provides the solution for a particular application. So, it looks like DP can be useful for applying to a wide variety of applications, but we shall soon see whether the algorithms based on this technique have good time complexities compared to those based on another technique.

Table 11.3. Comparisons of Time Complexity Among 4 Main Algorithm Design Techniques

Time Complexity			
	Towers of Hanoi	**0/1 Knapsack Problem**	**Shortest Paths in a Graph**
D&Q	$O(2^n)$ *		
BT		$O(2^N)$ **	
DP	$O(2^n)$	O(N*W) ***	Floyd Warshall Algorithm is based on All Pairs Shortest Path Problem $O(V^3)$ *****
GM			Dijkstra Algorithm is greedy O(E*log V) ****

* $O(2^n)$ where n is the number of disks

** $O(2^N)$ where N is the number of weight items

*** O(N*W) where N is the number of weight items and W is the capacity of the knapsack

**** O(E*log V) where E is the number of edges and V is the number of vertices in the graph.

***** $O(V^3)$ where V is the number of vertices in the graph.

As we can see in Towers of Hanoi application, both Divide and Conquer and Dynamic programming algorithms are good at solving the problem, and they both require $O(2^n)$ time, but one of them is still better than the other, which we will discuss in the next section.

The 0/1 Knapsack problem can be handled well by both backtracking and dynamic programming algorithms, but there is a trade-off between the two so that either one can be faster depending on the weight of the knapsack, which we will discuss in more detail in the next section.

Floyd-Warshall algorithm follows the dynamic programming (DP) paradigm while Dijkstra algorithm follows the greedy method (GP) approach. Dijkstra algorithm finds the shortest path from a single vertex source to all other vertices and the time complexity is O(E*log V). On the other hand, Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices. The time complexity here is $O(V^3)$.

We can use Dijkstra's shortest path algorithm for all pair shortest paths, but the time complexity here would be O(VE log V) which can go $O(V^3 \log V)$ in the worst case. We will discuss in more detail in the next section whether dynamic programming or greedy method yields the better solution for the concerned problem.

11.9 Discussion of Algorithmic Complexities for Particular Problems Using Algorithm Design Techniques

11.9.1 Towers of Hanoi Problem

Towers of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.

2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e., a disk can only be moved if it is the uppermost disk on a stack.

3) No disk can be placed on top of a smaller disk.

A solution to the Towers of Hanoi points to the recursive nature of divide and conquer strategy. It does more work on subproblems and and solves all of them as individual cases and then combine subproblems to the original problem, which requires more time consumption.

The time complexity of the Tower of Hanoi problem is $O(2^n)$, where n is the number of discs. This is because the divide and conquer algorithm recursively solves two subproblems of size n-1 at each step, and the number of steps required to solve a problem of size n is equal to 2^{n-1}.

A dynamic programming solution for Towers of Hanoi is the solution of the main problem that is expressed in terms of the solutions for the smaller overlapping subproblems and recursively defining the value of an optimal solution for each of these subproblems.

However, combining the solutions of these subproblems to solve the larger subproblems to reach an overall optimal solution by bottom-up approach is non-recursive. The DP technique solves subproblems only once and then stores them in a table to be reused so that the same subproblem will not have to be computed again.

Therefore, although the time complexity for both the technique approaches is $O(2^n)$, where n is the number of disks, dynamic programming technique is comparatively faster than divide and conquer strategy for Towers of Hanoi.

11.9.2 0/1 Knapsack Problem

A thief goes to a store to steal some items. There are multiple items available for different weights & profits.

Let us suppose there are 'N' number of items & weights of these are W_1, W_2,……W_N respectively, and the profits of these items are P_1, P_2,…….P_N respectively.

The thief wants to steal in such a way so that his overall profit be 'Maximum' and the 'Capacity constraint W' of the knapsack doesn't go out of limit.

In this 0/1 or binary knapsack problem, either a whole item is selected (1) or the whole item is not selected (0).

Here, the thief can't carry a fraction of the items.

Backtracking is similar to Dynamic Programming in that it solves a problem by efficiently performing an exhaustive search over the entire set of possible options. Backtracking is different in that it structures the search to be able to efficiently eliminate large sub-sets of solutions that are no longer possible.

The basic idea of 0/1 Knapsack using dynamic programming is to use a table to store the solutions of solved subproblems. If you face a subproblem again, you just need to take the solution in the table without having to solve it again. Therefore, the algorithms designed by dynamic programming are very effective. The time complexity for 0/1 Knapsack problem solved using DP is O(N*W) where N denotes number of items available and W denotes the capacity of the knapsack.

Usually in the backtracking approach, you continue to add items until W exceeds the maximum capacity of the knapsack and then we backtrack and make a recursive call. We don't consider this item for the "1" possibility and make another call for the particular item (in fact, we repeat the item) for '0' possibility and then we add further consecutive

items for the '1' possibility until W exceeds the maximum capacity of the knapsack. We repeat the process all the way up the tree. This makes the "recursive call tree" look like a binary tree which would have 2^N problems to solve (2^N nodes in that binary tree). So, the time complexity is $O(2^N)$.

Let us assume N=8, W=10000 where W has a larger value. Here N is polynomial in amount but the amount of W is non-polynomial.

When comparing this with the backtracking algorithm, it takes $O(2^N)$ time, but depending on W, either the dynamic programming algorithm is more efficient or the backtracking algorithm could be more efficient. For example, for N=8, W=100000, backtracking is preferable, but for N=30 and W=1000, the dynamic programming solution is preferable. Hence, there is a trade-off between the two.

11.9.3 Shortest Paths in a Graph Problem

As we have mentioned before, we can use Dijkstra's shortest path algorithm (which is based on the greedy method) for all pair shortest paths and it can go $O(V^3 \log V)$ in the worst case. On the other hand, Floyd-Warshall algorithm (which follows the dynamic programming approach) for the same problem has time complexity $O(V^3)$ in the best, average, and worst cases.

The three loops of Floyd-Warshall algorithm are clean compared to the not so clean three loops in the worst case of Dijkstra's algorithm. As a result, any case of Floyd algorithm is faster than the worst case of Dijkstra's algorithm.

```
n = no of vertices
A = matrix of dimension n*n
for k= 1 to n
  for i = 1 to n
    for j = 1 to n
      A^k[i, j] = min (A^k-1[i, j], A^k-1[i, k] + A^k-1[k, j]
  return A
```

Code 11.2: The Three Clean Loops of Floyd-Warshall Algorithm

As we can see in Code 11.2, there are three clean loops in Floyd-Warshall algorithm; nevertheless, the three loops in Dijkstra's algorithm in the worst case are not that clean for the all-pair shortest paths problem.

Summing up, this chapter made a survey about four main algorithm design techniques. We have found algorithms that commonly solve a problem, and even if they have the same time complexity, one is still better than the other.

We have also seen a case where they are trade-offs between the two and either one can be efficient under different constraints.

Also, we have observed that because one algorithm has clean nested loops while the other doesn't have not so clean ones, the former performs better with its time complexity being $O(V^3)$ compared to the latter with time complexity $O(V^3 \log V)$.

About the Author

Rosina S Khan has authored this book titled, "The Magical Guide to Algorithm Analysis and Design."

She has already published three other academic guides on free-ebooks.net which you will love to read. These are:

1) The Dummies' Guide to Database Systems: An Assembly of Information. An Easy-to-Understand Guide Even for Laypeople

2) The Dummies' Guide to Compiler Design

3) The Dummies' Guide to Software Engineering

She has taught courses in Computer Science and Engineering for many, many years as a faculty member in a private university in Dhaka city. She holds an M.Sc. degree in Software Technology from abroad.

Valuable Free Resources

- The author has authored an academic guide on Databases on free-ebooks.net, titled, "The Dummies' Guide to Database Systems: An Assembly of Information." https://www.free-ebooks.net/ebook/The-Dummies-Guide-to-Database-Systems-An-Assembly-of-Information

- The author has authored another academic guide on Compilers on free-ebooks.net titled, "The Dummies' Guide to Compiler Design." https://www.free-ebooks.net/computer-sciences-textbooks/The-Dummies-Guide-to-Compiler-Design

- The author has also authored an academic guide on Software Engineering on free-ebooks.net, "The Dummies' Guide to Software Engineering." https://www.free-ebooks.net/computer-sciences-textbooks/The-Dummies-Guide-to-Software-Engineering

- For a wealth of free resources based on stunning fiction stories (on free-ebooks.net), amazing self-help eBooks (self-published), commendable articles, quality scholar papers and valuable blogs, all authored by her, and much more, visit: https://www.rosinaskhan.weebly.com
  You will be glad that you did.

- You are also welcome to visit her Facebook page which shows a collection of her books based on fiction stories, self-help books and academic guides, along with large book images, which you can download for free as well as view inspirational quotes and online glamor articles. Here is the link:
  https://www.facebook.com/RosinaSKhan.hub

  Remember to like her page.

----------x----------